1. Introduction. BibTeX is a preprocessor (with elements of postprocessing as explained below) for the LaTeX document-preparation system. It handles most of the formatting decisions required to produce a reference list, outputting a `.bbl` file that a user can edit to add any finishing touches BibTeX isn't designed to handle (in practice, such editing almost never is needed); with this file LaTeX actually produces the reference list.

Here's how BibTeX works. It takes as input (a) an `.aux` file produced by LaTeX on an earlier run; (b) a `.bst` file (the style file), which specifies the general reference-list style and specifies how to format individual entries, and which is written by a style designer (called a wizard throughout this program) in a special-purpose language described in the BibTeX documentation—see the file `btxdoc.tex`; and (c) `.bib` file(s) constituting a database of all reference-list entries the user might ever hope to use. BibTeX chooses from the `.bib` file(s) only those entries specified by the `.aux` file (that is, those given by LaTeX's `\cite` or `\nocite` commands), and creates as output a `.bbl` file containing these entries together with the formatting commands specified by the `.bst` file (BibTeX also creates a `.blg` log file, which includes any error or warning messages, but this file isn't used by any program). LaTeX will use the `.bbl` file, perhaps edited by the user, to produce the reference list.

Many modules of BibTeX were taken from Knuth's TeX and TeXware, with his permission. All known system-dependent modules are marked in the index entry "system dependencies"; Dave Fuchs helped exorcise unwanted ones. In addition, a few modules that can be changed to make BibTeX smaller are marked in the index entry "space savings".

Megathanks to Howard Trickey, for whose suggestions future users and style writers would be eternally grateful, if only they knew.

The *banner* string defined here should be changed whenever BibTeX gets modified.

define *banner* ≡ `´This␣is␣BibTeX,␣Version␣0.99d´` { printed when the program starts }

2. Terminal output goes to the file *term_out*, while terminal input comes from *term_in*. On our system, these (system-dependent) files are already opened at the beginning of the program, and have the same real name.

define *term_out* ≡ *tty*
define *term_in* ≡ *tty*

3. This program uses the term *print* instead of *write* when writing on both the *log_file* and (system-dependent) *term_out* file, and it uses *trace_pr* when in **trace** mode, for which it writes on just the *log_file*. If you want to change where either set of macros writes to, you should also change the other macros in this program for that set; each such macro begins with *print_* or *trace_pr_*.

```
define print(#) ≡
        begin write(log_file,#); write(term_out,#);
        end
define print_ln(#) ≡
        begin write_ln(log_file,#); write_ln(term_out,#);
        end
define print_newline ≡ print_a_newline   { making this a procedure saves a little space }
define trace_pr(#) ≡
        begin write(log_file,#);
        end
define trace_pr_ln(#) ≡
        begin write_ln(log_file,#);
        end
define trace_pr_newline ≡
        begin write_ln(log_file);
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ ≡

```
procedure print_a_newline;
  begin write_ln(log_file); write_ln(term_out);
  end;
```

See also sections 18, 44, 45, 46, 47, 51, 53, 59, 82, 95, 96, 98, 99, 108, 111, 112, 113, 114, 115, 121, 128, 137, 138, 144, 148, 149, 150, 153, 157, 158, 159, 165, 166, 167, 168, 169, 188, 220, 221, 222, 226, 229, 230, 231, 232, 233, 234, 235, 240, 271, 280, 281, 284, 293, 294, 295, 310, 311, 313, 321, 356, 368, 373·$10^T s$456.

This code is used in section 12.

4. Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when BIBTEX is being installed or when system wizards are fooling around with BIBTEX without quite knowing what they are doing. Such code will not normally be compiled; it is delimited by the codewords '**debug**...**gubed**', with apologies to people who wish to preserve the purity of English. Similarly, there is some conditional code delimited by '**stat**...**tats**' that is intended only for use when statistics are to be kept about BIBTEX's memory/cpu usage, and there is conditional code delimited by '**trace**...**ecart**' that is intended to be a trace facility for use mainly when debugging `.bst` files.

```
define debug ≡ @{   { remove the '@{' when debugging }
define gubed ≡ @}   { remove the '@}' when debugging }
format debug ≡ begin
format gubed ≡ end

define stat ≡ @{   { remove the '@{' when keeping statistics }
define tats ≡ @}   { remove the '@}' when keeping statistics }
format stat ≡ begin
format tats ≡ end

define trace ≡ @{   { remove the '@{' when in trace mode }
define ecart ≡ @}   { remove the '@}' when in trace mode }
format trace ≡ begin
format ecart ≡ end
```

5. We assume that **case** statements may include a default case that applies if no matching label is found, since most PASCAL compilers have plugged this hole in the language by incorporating some sort of default mechanism. For example, the PASCAL-H compiler allows '*others*:' as a default label, and other PASCALs allow syntaxes like '**else**' or '*otherwise*' or '*otherwise*:', etc. The definitions of **othercases** and **endcases** should be changed to agree with local conventions. Note that no semicolon appears before **endcases** in this program, so the definition of **endcases** should include a semicolon if the compiler wants one. (Of course, if no default mechanism is available, the **case** statements of BIBTEX will have to be laboriously extended by listing all remaining cases. People who are stuck with such PASCALs have in fact done this, successfully but not happily!)

define *othercases* ≡ *others*: { default for cases not listed explicitly }
define *endcases* ≡ **end** { follows the default case in an extended **case** statement }
format *othercases* ≡ *else*
format *endcases* ≡ *end*

6. Labels are given symbolic names by the following definitions, so that occasional **goto** statements will be meaningful. We insert the label '*exit*:' just before the '**end**' of a procedure in which we have used the '**return**' statement defined below (and this is the only place '*exit*:' appears). This label is sometimes used for exiting loops that are set up with the **loop** construction defined below. Another generic label is '*loop_exit*:'; it appears immediately after a loop.

Incidentally, this program never declares a label that isn't actually used, because some fussy PASCAL compilers will complain about redundant labels.

define *exit* = 10 { go here to leave a procedure }
define *loop_exit* = 15 { go here to leave a loop within a procedure }
define *loop1_exit* = 16 { the first generic label for a procedure with two }
define *loop2_exit* = 17 { the second }

7. And **while** we're discussing loops: This program makes into **while** loops many that would otherwise be **for** loops because of Standard PASCAL limitations (it's a bit complicated—standard PASCAL doesn't allow a global variable as the index of a **for** loop inside a procedure; furthermore, many compilers have fairly severe limitations on the size of a block, including the main block of the program; so most of the code in this program occurs inside procedures, and since for other reasons this program must use primarily global variables, it doesn't use many **for** loops).

8. This program uses this convention: If there are several quantities in a boolean expression, they are ordered by expected frequency (except perhaps when an error message results) so that execution will be fastest; this is more an attempt to understand the program than to make it faster.

9. Here are some macros for common programming idioms.

define *incr*(#) ≡ # ← # + 1 { increase a variable by unity }
define *decr*(#) ≡ # ← # − 1 { decrease a variable by unity }
define *loop* ≡ **while** *true* **do** { repeat over and over until a **goto** happens }
format *loop* ≡ *xclause* { WEB's **xclause** acts like '**while** *true* **do**' }
define *do_nothing* ≡ { empty statement }
define *return* ≡ **goto** *exit* { terminate a procedure call }
format *return* ≡ *nil*
define *empty* = 0 { symbolic name for a null constant }
define *any_value* = 0 { this appeases PASCAL's boolean-evaluation scheme }

10. The main program. This program first reads the `.aux` file that LaTeX produces, (i) determining which `.bib` file(s) and `.bst` file to read and (ii) constructing a list of cite keys in order of occurrence. The `.aux` file may have other `.aux` files nested within. Second, it reads and executes the `.bst` file, (i) determining how and in which order to process the database entries in the `.bib` file(s) corresponding to those cite keys in the list (or in some cases, to all the entries in the `.bib` file(s)), (ii) determining what text to be output for each entry and determining any additional text to be output, and (iii) actually outputting this text to the `.bbl` file. In addition, the program sends error messages and other remarks to the *log_file* and terminal.

```
define close_up_shop = 9998  { jump here after fatal errors }
define exit_program = 9999  { jump here if we couldn't even get started }
⟨ Compiler directives 11 ⟩
program BibTeX;  { all files are opened dynamically }
  label close_up_shop, exit_program ⟨ Labels in the outer block 109 ⟩;
  const ⟨ Constants in the outer block 14 ⟩
  type ⟨ Types in the outer block 22 ⟩
  var ⟨ Globals in the outer block 16 ⟩
    ⟨ Procedures and functions for about everything 12 ⟩
    ⟨ The procedure initialize 13 ⟩
  begin initialize; print_ln(banner);
    ⟨ Read the .aux file 110 ⟩;
    ⟨ Read and execute the .bst file 151 ⟩;
  close_up_shop: ⟨ Clean up and leave 455 ⟩;
  exit_program: end.
```

11. If the first character of a PASCAL comment is a dollar sign, PASCAL-H treats the comment as a list of "compiler directives" that will affect the translation of this program into machine language. The directives shown below specify full checking and inclusion of the PASCAL debugger when BIBTEX is being debugged, but they cause range checking and other redundant code to be eliminated when the production system is being generated. Arithmetic overflow will be detected in all cases.

```
⟨ Compiler directives 11 ⟩ ≡
  @{@&$C−, A+, D−@}  { no range check, catch arithmetic overflow, no debug overhead }
  debug @{@&$C+, D+@} gubed  { but turn everything on when debugging }
```

This code is used in section 10.

12. All procedures in this program (except for *initialize*) are grouped into one of the seven classes below, and these classes are dispersed throughout the program. However: Much of this program is written top down, yet PASCAL wants its procedures bottom up. Since mooning is neither a technically nor a socially acceptable solution to the bottom-up problem, this section instead performs the topological gymnastics that WEB allows, ordering these classes to satisfy PASCAL compilers. There are a few procedures still out of place after this ordering, though, and the other modules that complete the task have "gymnastics" as an index entry.

```
⟨ Procedures and functions for about everything 12 ⟩ ≡
  ⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩
  ⟨ Procedures and functions for file-system interacting 38 ⟩
  ⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩
  ⟨ Procedures and functions for input scanning 83 ⟩
  ⟨ Procedures and functions for name-string processing 367 ⟩
  ⟨ Procedures and functions for style-file function execution 307 ⟩
  ⟨ Procedures and functions for the reading and processing of input files 100 ⟩
```

This code is used in section 10.

13. This procedure gets things started properly.

```
⟨ The procedure initialize 13 ⟩ ≡
procedure initialize;
  var ⟨ Local variables for initialization 23 ⟩
    begin ⟨ Check the "constant" values for consistency 17 ⟩;
    if (bad > 0) then
      begin write_ln(term_out, bad : 0, ´␣is␣a␣bad␣bad´); goto exit_program;
      end;
    ⟨ Set initial values of key variables 20 ⟩;
    pre_def_certain_strings;
    get_the_top_level_aux_file_name;
    end;
```

This code is used in section 10.

14. These parameters can be changed at compile time to extend or reduce BIBTEX's capacity. They are set to accommodate about 750 cites when used with the standard styles, although *pool_size* is usually the first limitation to be a problem, often when there are 500 cites.

```
⟨ Constants in the outer block 14 ⟩ ≡
  buf_size = 1000;  { maximum number of characters in an input line (or string) }
  min_print_line = 3;  { minimum .bbl line length: must be ≥ 3 }
  max_print_line = 79;  { the maximum: must be > min_print_line and < buf_size }
  aux_stack_size = 20;  { maximum number of simultaneous open .aux files }
  max_bib_files = 20;  { maximum number of .bib files allowed }
  pool_size = 65000;  { maximum number of characters in strings }
  max_strings = 4000;  { maximum number of strings, including pre-defined; must be ≤ hash_size }
  max_cites = 750;  { maximum number of distinct cite keys; must be ≤ max_strings }
  min_crossrefs = 2;  { minimum number of cross-refs required for automatic cite_list inclusion }
  wiz_fn_space = 3000;  { maximum amount of wiz_defined-function space }
  single_fn_space = 100;  { maximum amount for a single wiz_defined-function }
  max_ent_ints = 3000;  { maximum number of int_entry_vars (entries × int_entry_vars) }
  max_ent_strs = 3000;  { maximum number of str_entry_vars (entries × str_entry_vars) }
  ent_str_size = 100;  { maximum size of a str_entry_var; must be ≤ buf_size }
  glob_str_size = 1000;  { maximum size of a str_global_var; must be ≤ buf_size }
  max_fields = 17250;  { maximum number of fields (entries × fields, about 23 ∗ max_cites for consistency) }
  lit_stk_size = 100;  { maximum number of literal functions on the stack }
```

See also section 333.

This code is used in section 10.

15. These parameters can also be changed at compile time, but they're needed to define some WEB numeric macros so they must be so defined themselves.

```
define hash_size = 5000  { must be ≥ max_strings and ≥ hash_prime }
define hash_prime = 4253  { a prime number about 85% of hash_size and ≥ 128 and < 2^14 − 2^6 }
define file_name_size = 40  { file names shouldn't be longer than this }
define max_glob_strs = 10  { maximum number of str_global_var names }
define max_glb_str_minus_1 = max_glob_strs − 1  { to avoid wasting a str_global_var }
```

16. In case somebody has inadvertently made bad settings of the "constants," BIBTEX checks them using a global variable called *bad*.

This is the first of many sections of BIBTEX where global variables are defined.

⟨ Globals in the outer block 16 ⟩ ≡
bad: *integer*; { is some "constant" wrong? }

See also sections 19, 24, 30, 34, 37, 41, 43, 48, 65, 74, 76, 78, 80, 89, 91, 97, 104, 117, 124, 129, 147, 161, 163, 195, 219, 247, 290, 331, 337, 344·$10^T s$365.

This code is used in section 10.

17. Each digit-value of *bad* has a specific meaning.

⟨ Check the "constant" values for consistency 17 ⟩ ≡
bad ← 0;
if (*min_print_line* < 3) **then** *bad* ← 1;
if (*max_print_line* ≤ *min_print_line*) **then** *bad* ← 10 ∗ *bad* + 2;
if (*max_print_line* ≥ *buf_size*) **then** *bad* ← 10 ∗ *bad* + 3;
if (*hash_prime* < 128) **then** *bad* ← 10 ∗ *bad* + 4;
if (*hash_prime* > *hash_size*) **then** *bad* ← 10 ∗ *bad* + 5;
if (*hash_prime* ≥ (16384 − 64)) **then** *bad* ← 10 ∗ *bad* + 6;
if (*max_strings* > *hash_size*) **then** *bad* ← 10 ∗ *bad* + 7;
if (*max_cites* > *max_strings*) **then** *bad* ← 10 ∗ *bad* + 8;
if (*ent_str_size* > *buf_size*) **then** *bad* ← 10 ∗ *bad* + 9;
if (*glob_str_size* > *buf_size*) **then** *bad* ← 100 ∗ *bad* + 11; { well, almost each }

See also section 302.

This code is used in section 13.

18. A global variable called *history* will contain one of four values at the end of every run: *spotless* means that no unusual messages were printed; *warning_message* means that a message of possible interest was printed but no serious errors were detected; *error_message* means that at least one error was found; *fatal_message* means that the program terminated abnormally. The value of *history* does not influence the behavior of the program; it is simply computed for the convenience of systems that might want to use such information.

```
define spotless = 0  { history value for normal jobs }
define warning_message = 1  { history value when non-serious info was printed }
define error_message = 2  { history value when an error was noted }
define fatal_message = 3  { history value when we had to stop prematurely }
```

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure mark_warning;
  begin if (history = warning_message) then incr(err_count)
  else if (history = spotless) then
      begin history ← warning_message; err_count ← 1;
      end;
  end;
procedure mark_error;
  begin if (history < error_message) then
    begin history ← error_message; err_count ← 1;
    end
  else  { history = error_message }
  incr(err_count);
  end;
procedure mark_fatal;
  begin history ← fatal_message;
  end;
```

19. For the two states *warning_message* and *error_message* we keep track of the number of messages given; but since *warning_message*s aren't so serious, we ignore them once we've seen an *error_message*. Hence we need just the single variable *err_count* to keep track.

```
⟨ Globals in the outer block 16 ⟩ +≡
history: spotless .. fatal_message;  { how bad was this run? }
err_count: integer;
```

20. The *err_count* gets set or reset when *history* first changes to *warning_message* or *error_message*, so we don't need to initialize it.

```
⟨ Set initial values of key variables 20 ⟩ ≡
  history ← spotless;
```

See also sections 25, 27, 28, 32, 33, 35, 67, 72, 119, 125, 131, 162, 164, 196·10^T *s*292.

This code is used in section 13.

21. The character set. (The following material is copied (almost) verbatim from TEX. Thus, the same system-dependent changes should be made to both programs.)

In order to make TEX readily portable between a wide variety of computers, all of its input text is converted to an internal seven-bit code that is essentially standard ASCII, the "American Standard Code for Information Interchange." This conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user's external representation just before they are output to a text file.

Such an internal code is relevant to users of TEX primarily because it governs the positions of characters in the fonts. For example, the character 'A' has ASCII code 65 = *'101*, and when TEX typesets this letter it specifies character number 65 in the current font. If that font actually has 'A' in a different position, TEX doesn't know what the real position is; the program that does the actual printing from TEX's device-independent files is responsible for converting from ASCII to a particular font encoding.

TEX's internal code is relevant also with respect to constants that begin with a reverse apostrophe.

22. Characters of text that have been converted to TEX's internal form are said to be of type *ASCII_code*, which is a subrange of the integers.

⟨ Types in the outer block 22 ⟩ ≡
  ASCII_code = 0 .. 127; { seven-bit numbers }

See also sections 31, 36, 42, 49, 64, 73, 105, 118, 130, 160, 291·$10^T s332$.

This code is used in section 10.

23. The original PASCAL compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower-case letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for typesetting; so the present specification of TEX has been written under the assumption that the PASCAL compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes *'40* through *'176*; all of these characters are now available on most computer terminals.

Since we are dealing with more characters than were present in the first PASCAL compilers, we have to decide what to call the associated data type. Some PASCALs use the original name *char* for the characters in text files, even though there now are more than 64 such characters, while other PASCALs consider *char* to be a 64-element subrange of a larger data type that has some other name.

In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters that are converted to and from *ASCII_code* when they are input and output. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *text_char* ≡ *char* { the data type of characters in text files }
define *first_text_char* = 0 { ordinal number of the smallest element of *text_char* }
define *last_text_char* = 127 { ordinal number of the largest element of *text_char* }

⟨ Local variables for initialization 23 ⟩ ≡
i: 0 .. *last_text_char*; { this is the first one declared }

See also section 66.

This code is used in section 13.

24. The TEX processor converts between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to PASCAL's *ord* and *chr* functions.

⟨ Globals in the outer block 16 ⟩ +≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }

25. Since we are assuming that our PASCAL system is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize most of the *xchr* array properly, without needing any system-dependent changes. On the other hand, it is possible to implement TeX with less complete character sets, and in such cases it will be necessary to change something here.

⟨ Set initial values of key variables 20 ⟩ +≡

```
xchr['40] ← ' '; xchr['41] ← '!'; xchr['42] ← '"'; xchr['43] ← '#'; xchr['44] ← '$';
xchr['45] ← '%'; xchr['46] ← '&'; xchr['47] ← ''';
xchr['50] ← '('; xchr['51] ← ')'; xchr['52] ← '*'; xchr['53] ← '+'; xchr['54] ← ',';
xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
xchr['60] ← '0'; xchr['61] ← '1'; xchr['62] ← '2'; xchr['63] ← '3'; xchr['64] ← '4';
xchr['65] ← '5'; xchr['66] ← '6'; xchr['67] ← '7';
xchr['70] ← '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] ← ';'; xchr['74] ← '<';
xchr['75] ← '='; xchr['76] ← '>'; xchr['77] ← '?';
xchr['100] ← '@'; xchr['101] ← 'A'; xchr['102] ← 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
xchr['105] ← 'E'; xchr['106] ← 'F'; xchr['107] ← 'G';
xchr['110] ← 'H'; xchr['111] ← 'I'; xchr['112] ← 'J'; xchr['113] ← 'K'; xchr['114] ← 'L';
xchr['115] ← 'M'; xchr['116] ← 'N'; xchr['117] ← 'O';
xchr['120] ← 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] ← 'S'; xchr['124] ← 'T';
xchr['125] ← 'U'; xchr['126] ← 'V'; xchr['127] ← 'W';
xchr['130] ← 'X'; xchr['131] ← 'Y'; xchr['132] ← 'Z'; xchr['133] ← '['; xchr['134] ← '\';
xchr['135] ← ']'; xchr['136] ← '^'; xchr['137] ← '_';
xchr['140] ← '`'; xchr['141] ← 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
xchr['150] ← 'h'; xchr['151] ← 'i'; xchr['152] ← 'j'; xchr['153] ← 'k'; xchr['154] ← 'l';
xchr['155] ← 'm'; xchr['156] ← 'n'; xchr['157] ← 'o';
xchr['160] ← 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] ← 's'; xchr['164] ← 't';
xchr['165] ← 'u'; xchr['166] ← 'v'; xchr['167] ← 'w';
xchr['170] ← 'x'; xchr['171] ← 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; xchr['174] ← '|';
xchr['175] ← '}'; xchr['176] ← '~';
xchr[0] ← ' '; xchr['177] ← ' ';   { ASCII codes 0 and '177 do not appear in text }
```

26. Some of the ASCII codes without visible characters have been given symbolic names in this program because they are used with a special meaning. The *tab* character may be system dependent.

```
define null_code = '0   { ASCII code that might disappear }
define tab = '11   { ASCII code treated as white_space }
define space = '40   { ASCII code treated as white_space }
define invalid_code = '177   { ASCII code that should not appear }
```

27. The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. Appendix C of *The TEXbook* gives a complete specification of the intended correspondence between characters and TEX's internal representation.

If TEX is being used on a garden-variety PASCAL for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in $xchr[1\,..\,'37]$, but the safest policy is to blank everything out by using the code shown below.

However, other settings of *xchr* will make TEX more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '`\ne`'. At MIT, for example, it would be more appropriate to substitute the code

$$\textbf{for } i \leftarrow 1 \textbf{ to } '37 \textbf{ do } xchr[i] \leftarrow chr(i);$$

TEX's character set is essentially the same as MIT's, even with respect to characters less than $'40$. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of TEX are allowed to have in their input files. It is best to make the codes correspond to the intended interpretations as shown in Appendix C whenever possible; but this is not necessary. For example, in countries with an alphabet of more than 26 letters, it is usually best to map the additional letters into codes less than $'40$.

⟨ Set initial values of key variables 20 ⟩ +≡
```
for i ← 1 to '37 do xchr[i] ← '␣';
xchr[tab] ← chr(tab);
```

28. This system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*. Note that if $xchr[i] = xchr[j]$ where $i < j < '177$, the value of $xord[xchr[i]]$ will turn out to be j or more; hence, standard ASCII code numbers will be used instead of codes below $'40$ in case there is a coincidence.

⟨ Set initial values of key variables 20 ⟩ +≡
```
for i ← first_text_char to last_text_char do xord[chr(i)] ← invalid_code;
for i ← 1 to '176 do xord[xchr[i]] ← i;
```

29. Also, various characters are given symbolic names; all the ones this program uses are collected here. We use the sharp sign as the *concat_char*, rather than something more natural (like an ampersand), for uniformity of database syntax (ampersand is a valid character in identifiers).

define *double_quote* = `"""` { delimits strings }
define *number_sign* = `"#"` { marks an *int_literal* }
define *comment* = `"%"` { ignore the rest of a `.bst` or TeX line }
define *single_quote* = `"'"` { marks a quoted function }
define *left_paren* = `"("` { optional database entry left delimiter }
define *right_paren* = `")"` { corresponding right delimiter }
define *comma* = `","` { separates various things }
define *minus_sign* = `"-"` { for a negative number }
define *equals_sign* = `"="` { separates a field name from a field value }
define *at_sign* = `"@"` { the beginning of a database entry }
define *left_brace* = `"{"` { left delimiter of many things }
define *right_brace* = `"}"` { corresponding right delimiter }
define *period* = `"."` { these are three }
define *question_mark* = `"?"` { string-ending characters }
define *exclamation_mark* = `"!"` { of interest in `add.period$` }
define *tie* = `"~"` { the default space char, in `format.name$` }
define *hyphen* = `"-"` { like *white_space*, in `format.name$` }
define *star* = `"*"` { for including entire database }
define *concat_char* = `"#"` { for concatenating field tokens }
define *colon* = `":"` { for lower-casing (usually title) strings }
define *backslash* = `"\"` { used to recognize accented characters }

30. These arrays give a lexical classification for the *ASCII_code*s; *lex_class* is used for general scanning and *id_class* is used for scanning identifiers.

⟨ Globals in the outer block 16 ⟩ +≡
lex_class: **array** [*ASCII_code*] **of** *lex_type*;
id_class: **array** [*ASCII_code*] **of** *id_type*;

31. Every character has two types of the lexical classifications. The first type is general, and the second type tells whether the character is legal in identifiers.

define *illegal* = 0 { the unrecognized *ASCII_code*s }
define *white_space* = 1 { things like *space*s that you can't see }
define *alpha* = 2 { the upper- and lower-case letters }
define *numeric* = 3 { the ten digits }
define *sep_char* = 4 { things sometimes treated like *white_space* }
define *other_lex* = 5 { when none of the above applies }
define *last_lex* = 5 { the same number as on the line above }

define *illegal_id_char* = 0 { a few forbidden ones }
define *legal_id_char* = 1 { most printing characters }

⟨ Types in the outer block 22 ⟩ +≡
lex_type = 0 .. *last_lex*;
id_type = 0 .. 1;

32. Now we initialize the system-dependent *lex_class* array. The *tab* character may be system dependent. Note that the order of these assignments is important here.

⟨ Set initial values of key variables 20 ⟩ +≡
```
for i ← 0 to ´177 do lex_class[i] ← other_lex;
for i ← 0 to ´37 do lex_class[i] ← illegal;
lex_class[invalid_code] ← illegal; lex_class[tab] ← white_space; lex_class[space] ← white_space;
lex_class[tie] ← sep_char; lex_class[hyphen] ← sep_char;
for i ← ´60 to ´71 do lex_class[i] ← numeric;
for i ← ´101 to ´132 do lex_class[i] ← alpha;
for i ← ´141 to ´172 do lex_class[i] ← alpha;
```

33. And now the *id_class* array.

⟨ Set initial values of key variables 20 ⟩ +≡
```
for i ← 0 to ´177 do id_class[i] ← legal_id_char;
for i ← 0 to ´37 do id_class[i] ← illegal_id_char;
id_class[space] ← illegal_id_char; id_class[tab] ← illegal_id_char; id_class[double_quote] ← illegal_id_char;
id_class[number_sign] ← illegal_id_char; id_class[comment] ← illegal_id_char;
id_class[single_quote] ← illegal_id_char; id_class[left_paren] ← illegal_id_char;
id_class[right_paren] ← illegal_id_char; id_class[comma] ← illegal_id_char;
id_class[equals_sign] ← illegal_id_char; id_class[left_brace] ← illegal_id_char;
id_class[right_brace] ← illegal_id_char;
```

34. The array *char_width* gives relative printing widths of each *ASCII_code*, and *string_width* will be used later to sum up *char_width*s in a string.

⟨ Globals in the outer block 16 ⟩ +≡
```
char_width: array [ASCII_code] of integer;
string_width: integer;
```

35. Now we initialize the system-dependent *char_width* array, for which *space* is the only *white_space* character given a nonzero printing width. The widths here are taken from Stanford's June '87 *cmr*10 font and represent hundredths of a point (rounded), but since they're used only for relative comparisons, the units have no meaning.

define *ss_width* = 500 { character *'31*'s width in the *cmr*10 font }
define *ae_width* = 722 { character *'32*'s width in the *cmr*10 font }
define *oe_width* = 778 { character *'33*'s width in the *cmr*10 font }
define *upper_ae_width* = 903 { character *'35*'s width in the *cmr*10 font }
define *upper_oe_width* = 1014 { character *'36*'s width in the *cmr*10 font }

⟨ Set initial values of key variables 20 ⟩ +≡
for $i \leftarrow 0$ **to** *'177* **do** *char_width*[i] ← 0;
char_width[*'40*] ← 278; *char_width*[*'41*] ← 278; *char_width*[*'42*] ← 500; *char_width*[*'43*] ← 833;
char_width[*'44*] ← 500; *char_width*[*'45*] ← 833; *char_width*[*'46*] ← 778; *char_width*[*'47*] ← 278;
char_width[*'50*] ← 389; *char_width*[*'51*] ← 389; *char_width*[*'52*] ← 500; *char_width*[*'53*] ← 778;
char_width[*'54*] ← 278; *char_width*[*'55*] ← 333; *char_width*[*'56*] ← 278; *char_width*[*'57*] ← 500;
char_width[*'60*] ← 500; *char_width*[*'61*] ← 500; *char_width*[*'62*] ← 500; *char_width*[*'63*] ← 500;
char_width[*'64*] ← 500; *char_width*[*'65*] ← 500; *char_width*[*'66*] ← 500; *char_width*[*'67*] ← 500;
char_width[*'70*] ← 500; *char_width*[*'71*] ← 500; *char_width*[*'72*] ← 278; *char_width*[*'73*] ← 278;
char_width[*'74*] ← 278; *char_width*[*'75*] ← 778; *char_width*[*'76*] ← 472; *char_width*[*'77*] ← 472;
char_width[*'100*] ← 778; *char_width*[*'101*] ← 750; *char_width*[*'102*] ← 708; *char_width*[*'103*] ← 722;
char_width[*'104*] ← 764; *char_width*[*'105*] ← 681; *char_width*[*'106*] ← 653; *char_width*[*'107*] ← 785;
char_width[*'110*] ← 750; *char_width*[*'111*] ← 361; *char_width*[*'112*] ← 514; *char_width*[*'113*] ← 778;
char_width[*'114*] ← 625; *char_width*[*'115*] ← 917; *char_width*[*'116*] ← 750; *char_width*[*'117*] ← 778;
char_width[*'120*] ← 681; *char_width*[*'121*] ← 778; *char_width*[*'122*] ← 736; *char_width*[*'123*] ← 556;
char_width[*'124*] ← 722; *char_width*[*'125*] ← 750; *char_width*[*'126*] ← 750; *char_width*[*'127*] ← 1028;
char_width[*'130*] ← 750; *char_width*[*'131*] ← 750; *char_width*[*'132*] ← 611; *char_width*[*'133*] ← 278;
char_width[*'134*] ← 500; *char_width*[*'135*] ← 278; *char_width*[*'136*] ← 500; *char_width*[*'137*] ← 278;
char_width[*'140*] ← 278; *char_width*[*'141*] ← 500; *char_width*[*'142*] ← 556; *char_width*[*'143*] ← 444;
char_width[*'144*] ← 556; *char_width*[*'145*] ← 444; *char_width*[*'146*] ← 306; *char_width*[*'147*] ← 500;
char_width[*'150*] ← 556; *char_width*[*'151*] ← 278; *char_width*[*'152*] ← 306; *char_width*[*'153*] ← 528;
char_width[*'154*] ← 278; *char_width*[*'155*] ← 833; *char_width*[*'156*] ← 556; *char_width*[*'157*] ← 500;
char_width[*'160*] ← 556; *char_width*[*'161*] ← 528; *char_width*[*'162*] ← 392; *char_width*[*'163*] ← 394;
char_width[*'164*] ← 389; *char_width*[*'165*] ← 556; *char_width*[*'166*] ← 528; *char_width*[*'167*] ← 722;
char_width[*'170*] ← 528; *char_width*[*'171*] ← 528; *char_width*[*'172*] ← 444; *char_width*[*'173*] ← 500;
char_width[*'174*] ← 1000; *char_width*[*'175*] ← 500; *char_width*[*'176*] ← 500;

36. Input and output. The basic operations we need to do are (1) inputting and outputting of text characters to or from a file; (2) instructing the operating system to initiate ("open") or to terminate ("close") input or output to or from a specified file; and (3) testing whether the end of an input file has been reached.

⟨ Types in the outer block 22 ⟩ +≡
  alpha_file = **packed file of** *text_char*; { files that contain textual data }

37. Most of what we need to do with respect to input and output can be handled by the I/O facilities that are standard in PASCAL, i.e., the routines called *get*, *put*, *eof*, and so on. But standard PASCAL does not allow file variables to be associated with file names that are determined at run time, so it cannot be used to implement BIBTEX; some sort of extension to PASCAL's ordinary *reset* and *rewrite* is crucial for our purposes. We shall assume that *name_of_file* is a variable of an appropriate type such that the PASCAL run-time system being used to implement BIBTEX can open a file whose external name is specified by *name_of_file*. BIBTEX does no case conversion for file names.

⟨ Globals in the outer block 16 ⟩ +≡
name_of_file: **packed array** [1 .. *file_name_size*] **of** *char*; { on some systems this is a **record** variable }
name_length: 0 .. *file_name_size*; { this many characters are relevant in *name_of_file* (the rest are blank) }
name_ptr: 0 .. *file_name_size* + 1; { index variable into *name_of_file* }

38. The PASCAL-H compiler with which the present version of TEX was prepared has extended the rules of PASCAL in a very convenient way. To open file f, we can write

reset(f, *name*, ´/O´) for input;
rewrite(f, *name*, ´/O´) for output.

The '*name*' parameter, which is of type '**packed array** [⟨*any*⟩] **of** *text_char*', stands for the name of the external file that is being opened for input or output. Blank spaces that might appear in *name* are ignored.

The '/O' parameter tells the operating system not to issue its own error messages if something goes wrong. If a file of the specified name cannot be found, or if such a file cannot be opened for some other reason (e.g., someone may already be trying to write the same file), we will have *erstat*(f) ≠ 0 after an unsuccessful *reset* or *rewrite*. This allows TEX to undertake appropriate corrective action.

TEX's file-opening procedures return *false* if no file identified by *name_of_file* could be opened.

define *reset_OK*(#) ≡ *erstat*(#) = 0
define *rewrite_OK*(#) ≡ *erstat*(#) = 0

⟨ Procedures and functions for file-system interacting 38 ⟩ ≡
function *erstat* (**var** f : **file**) : *integer*; *extern*; { in the runtime library }

function *a_open_in*(**var** f : *alpha_file*): *boolean*; { open a text file for input }
  begin *reset*(f, *name_of_file*, ´/O´); *a_open_in* ← *reset_OK*(f);
  end;
function *a_open_out*(**var** f : *alpha_file*): *boolean*; { open a text file for output }
  begin *rewrite*(f, *name_of_file*, ´/O´); *a_open_out* ← *rewrite_OK*(f);
  end;

See also sections 39, 58, 60·10^T s61.

This code is used in section 12.

39. Files can be closed with the PASCAL-H routine '*close*(*f*)', which should be used when all input or output with respect to *f* has been completed. This makes *f* available to be opened again, if desired; and if *f* was used for output, the *close* operation makes the corresponding external file appear on the user's area, ready to be read.

```
⟨ Procedures and functions for file-system interacting 38 ⟩ +≡
procedure a_close(var f : alpha_file); { close a text file }
  begin close(f);
  end;
```

40. Text output is easy to do with the ordinary PASCAL *put* procedure, so we don't have to make any other special arrangements. The treatment of text input is more difficult, however, because of the necessary translation to *ASCII_code* values, and because TEX's conventions should be efficient and they should blend nicely with the user's operating environment.

41. Input from text files is read one line at a time, using a routine called *input_ln*. This function is defined in terms of global variables called *buffer* and *last*. The *buffer* array contains *ASCII_code* values, and *last* is an index into this array marking the end of a line of text. (Occasionally, *buffer* is used for something else, in which case it is copied to a temporary array.)

```
⟨ Globals in the outer block 16 ⟩ +≡
buffer: buf_type; { usually, lines of characters being read }
last: buf_pointer; { end of the line just input to buffer }
```

42. The type *buf_type* is used for *buffer*, for saved copies of it, or for scratch work. It's not **packed** because otherwise the program would run much slower on some systems (more than 25 percent slower, for example, on a TOPS-20 operating system). But on systems that are byte-addressable and that have a good compiler, packing *buf_type* would save lots of space without much loss of speed. Other modules that have packable arrays are also marked with a "space savings" index entry.

```
⟨ Types in the outer block 22 ⟩ +≡
  buf_pointer = 0 .. buf_size; { an index into a buf_type }
  buf_type = array [buf_pointer] of ASCII_code; { for various buffers }
```

43. And while we're at it, we declare another buffer for general use. Because buffers are not packed and can get large, we use *sv_buffer* several purposes; this is a bit kludgy, but it helps make the stack space not overflow on some machines. It's used when reading the entire database file (in the `read` command) and when doing name-handling (through the alias *name_buf*) in the *built_in* functions `format.names$` and `num.names$`.

```
⟨ Globals in the outer block 16 ⟩ +≡
sv_buffer: buf_type;
sv_ptr1: buf_pointer;
sv_ptr2: buf_pointer;
tmp_ptr, tmp_end_ptr: integer; { copy pointers only, usually for buffers }
```

44. When something in the program wants to be bigger or something out there wants to be smaller, it's time to call it a run. Here's the first of several macros that have associated procedures so that they produce less inline code.

```
define overflow(#) ≡
        begin  { fatal error—close up shop }
        print_overflow; print_ln(#:0); goto close_up_shop;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure print_overflow;
  begin print('Sorry---you''ve␣exceeded␣BibTeX''s␣'); mark_fatal;
  end;
```

45. When something happens that the program thinks is impossible, call the maintainer.

```
define confusion(#) ≡
        begin  { fatal error—close up shop }
        print(#); print_confusion; goto close_up_shop;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure print_confusion;
  begin print_ln('---this␣can''t␣happen'); print_ln('*Please␣notify␣the␣BibTeX␣maintainer*');
  mark_fatal;
  end;
```

46. When a buffer overflows, it's time to complain (and then quit).

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure buffer_overflow;
  begin overflow('buffer␣size␣', buf_size);
  end;
```

47. The *input_ln* function brings the next line of input from the specified file into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets $last \leftarrow 0$. In general, the *ASCII_code* numbers that represent the next line of the file are input into $buffer[0]$, $buffer[1]$, ..., $buffer[last-1]$; and the global variable *last* is set equal to the length of the line. Trailing *white_space* characters are removed from the line (*white_space* characters are explained in the character-set section—most likely they're blanks); thus, either $last = 0$ (in which case the line was entirely blank) or $lex\_class[buffer[last-1]] \neq white\_space$. An overflow error is given if the normal actions of *input_ln* would make $last > buf\_size$.

Standard PASCAL says that a file should have *eoln* immediately before *eof*, but BibTeX needs only a weaker restriction: If *eof* occurs in the middle of a line, the system function *eoln* should return a *true* result (even though $f\uparrow$ will be undefined).

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
function input_ln(var f : alpha_file): boolean;  { inputs the next line or returns false }
  label loop_exit;
  begin last ← 0;
  if (eof(f)) then input_ln ← false
  else begin while (¬eoln(f)) do
      begin if (last ≥ buf_size) then buffer_overflow;
      buffer[last] ← xord[f↑]; get(f); incr(last);
      end;
    get(f);
    while (last > 0) do  { remove trailing white_space }
      if (lex_class[buffer[last − 1]] = white_space) then decr(last)
      else goto loop_exit;
  loop_exit: input_ln ← true;
    end;
  end;
```

48. String handling. BIBTEX uses variable-length strings of seven-bit characters. Since PASCAL does not have a well-developed string mechanism, BIBTEX does all its string processing by home-grown (predominantly TEX's) methods. Unlike TEX, however, BIBTEX does not use a *pool_file* for string storage; it creates its few pre-defined strings at run-time.

The necessary operations are handled with a simple data structure. The array *str_pool* contains all the (seven-bit) ASCII codes in all the strings BIBTEX must ever search for (generally identifiers names), and the array *str_start* contains indices of the starting points of each such string. Strings are referred to by integer numbers, so that string number s comprises the characters $str\_pool[j]$ for $str\_start[s] \le j < str\_start[s+1]$. Additional integer variables *pool_ptr* and *str_ptr* indicate the number of entries used so far in *str_pool* and *str_start*; locations $str\_pool[pool\_ptr]$ and $str\_start[str\_ptr]$ are ready for the next string to be allocated. Location $str\_start[0]$ is unused so that hashing will work correctly.

Elements of the *str_pool* array must be ASCII codes that can actually be printed; i.e., they must have an *xchr* equivalent in the local character set.

```
⟨ Globals in the outer block 16 ⟩ +≡
str_pool: packed array [pool_pointer] of ASCII_code;  { the characters }
str_start: packed array [str_number] of pool_pointer;  { the starting pointers }
pool_ptr: pool_pointer;  { first unused position in str_pool }
str_ptr: str_number;  { start of the current string being created }
str_num: str_number;  { general index variable into str_start }
p_ptr1, p_ptr2: pool_pointer;  { several procedures use these locally }
```

49. Where *pool_pointer* and *str_number* are pointers into *str_pool* and *str_start*.

```
⟨ Types in the outer block 22 ⟩ +≡
  pool_pointer = 0 .. pool_size;  { for variables that point into str_pool }
  str_number = 0 .. max_strings;  { for variables that point into str_start }
```

50. These macros send a string in *str_pool* to an output file.

```
define max_pop = 3  { —see the built_in functions section }
define print_pool_str(#) ≡ print_a_pool_str(#)  { making this a procedure saves a little space }
define trace_pr_pool_str(#) ≡
        begin out_pool_str(log_file, #);
        end
```

51. And here are the associated procedures. Note: The *term_out* file is system dependent.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure out_pool_str(var f : alpha_file; s : str_number);
  var i: pool_pointer;
  begin  { allowing str_ptr ≤ s < str_ptr + max_pop is a .bst-stack kludge }
  if ((s < 0) ∨ (s ≥ str_ptr + max_pop) ∨ (s ≥ max_strings)) then
    confusion('Illegal␣string␣number:', s : 0);
  for i ← str_start[s] to str_start[s + 1] − 1 do write(f, xchr[str_pool[i]]);
  end;
procedure print_a_pool_str(s : str_number);
  begin out_pool_str(term_out, s); out_pool_str(log_file, s);
  end;
```

52. Several of the elementary string operations are performed using WEB macros instead of using PASCAL procedures, because many of the operations are done quite frequently and we want to avoid the overhead of procedure calls. For example, here is a simple macro that computes the length of a string.

```
define length(#) ≡ (str_start[# + 1] − str_start[#])  { the number of characters in string number # }
```

53. Strings are created by appending character codes to *str_pool*. The macro called *append_char*, defined here, does not check to see if the value of *pool_ptr* has gotten too high; this test is supposed to be made before *append_char* is used.

To test if there is room to append l more characters to *str_pool*, we shall write *str_room*(l), which aborts BibTeX and gives an error message if there isn't enough room.

```
define append_char(#) ≡   { put ASCII_code # at the end of str_pool }
        begin str_pool[pool_ptr] ← #; incr(pool_ptr);
        end
define str_room(#) ≡   { make sure that the pool hasn't overflowed }
        begin if (pool_ptr + # > pool_size) then pool_overflow;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure pool_overflow;
  begin overflow('pool␣size␣', pool_size);
  end;
```

54. Once a sequence of characters has been appended to *str_pool*, it officially becomes a string when the function *make_string* is called. It returns the string number of the string it just made.

⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ ≡

```
function make_string: str_number;   { current string enters the pool }
  begin if (str_ptr = max_strings) then overflow('number␣of␣strings␣', max_strings);
  incr(str_ptr); str_start[str_ptr] ← pool_ptr; make_string ← str_ptr − 1;
  end;
```

See also sections 56, 57, 62, 63, 68, 77, 198, 265, 278, 300, 301, 303, 335·10^T*s*336.

This code is used in section 12.

55. These macros destroy and recreate the string at the end of the pool.

```
define flush_string ≡
        begin decr(str_ptr); pool_ptr ← str_start[str_ptr];
        end
define unflush_string ≡
        begin incr(str_ptr); pool_ptr ← str_start[str_ptr];
        end
```

56. This subroutine compares string s with another string that appears in the buffer *buf* between positions *bf_ptr* and *bf_ptr* + *len* − 1; the result is *true* if and only if the strings are equal.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
function str_eq_buf(s : str_number; var buf : buf_type; bf_ptr, len : buf_pointer): boolean;
        { test equality of strings }
  label exit;
  var i: buf_pointer;   { running }
    j: pool_pointer;   { indices }
  begin if (length(s) ≠ len) then   { strings of unequal length }
    begin str_eq_buf ← false; return;
    end;
  i ← bf_ptr; j ← str_start[s];
  while (j < str_start[s + 1]) do
    begin if (str_pool[j] ≠ buf[i]) then
      begin str_eq_buf ← false; return;
      end;
    incr(i); incr(j);
    end;
  str_eq_buf ← true;
exit: end;
```

57. This subroutine compares two *str_pool* strings and returns true *true* if and only if the strings are equal.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
function str_eq_str(s1, s2 : str_number): boolean;
  label exit;
  begin if (length(s1) ≠ length(s2)) then
    begin str_eq_str ← false; return;
    end;
  p_ptr1 ← str_start[s1]; p_ptr2 ← str_start[s2];
  while (p_ptr1 < str_start[s1 + 1]) do
    begin if (str_pool[p_ptr1] ≠ str_pool[p_ptr2]) then
      begin str_eq_str ← false; return;
      end;
    incr(p_ptr1); incr(p_ptr2);
    end;
  str_eq_str ← true;
exit: end;
```

58. This procedure copies file name *file_name* into the beginning of *name_of_file*, if it will fit. It also sets the global variable *name_length* to the appropriate value.

```
⟨ Procedures and functions for file-system interacting 38 ⟩ +≡
procedure start_name(file_name : str_number);
  var p_ptr: pool_pointer;  { running index }
  begin if (length(file_name) > file_name_size) then
    begin print('File=''); print_pool_str(file_name); print_ln(',''); file_nm_size_overflow;
    end;
  name_ptr ← 1; p_ptr ← str_start[file_name];
  while (p_ptr < str_start[file_name + 1]) do
    begin name_of_file[name_ptr] ← chr(str_pool[p_ptr]); incr(name_ptr); incr(p_ptr);
    end;
  name_length ← length(file_name);
  end;
```

59. Yet another complaint-before-quitting.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure file_nm_size_overflow;
  begin overflow('file␣name␣size␣'', file_name_size);
  end;
```

60. This procedure copies file extension *ext* into the array *name_of_file* starting at position *name_length*+1. It also sets the global variable *name_length* to the appropriate value.

```
⟨ Procedures and functions for file-system interacting 38 ⟩ +≡
procedure add_extension(ext : str_number);
  var p_ptr: pool_pointer;  { running index }
  begin if (name_length + length(ext) > file_name_size) then
    begin print('File=', name_of_file, ',␣extension='); print_pool_str(ext); print_ln(',');
    file_nm_size_overflow;
    end;
  name_ptr ← name_length + 1; p_ptr ← str_start[ext];
  while (p_ptr < str_start[ext + 1]) do
    begin name_of_file[name_ptr] ← chr(str_pool[p_ptr]); incr(name_ptr); incr(p_ptr);
    end;
  name_length ← name_length + length(ext); name_ptr ← name_length + 1;
  while (name_ptr ≤ file_name_size) do  { pad with blanks }
    begin name_of_file[name_ptr] ← '␣'; incr(name_ptr);
    end;
  end;
```

61. This procedure copies the default logical area name *area* into the array *name_of_file* starting at position 1, after shifting up the rest of the filename. It also sets the global variable *name_length* to the appropriate value.

```
⟨ Procedures and functions for file-system interacting 38 ⟩ +≡
procedure add_area(area : str_number);
  var p_ptr: pool_pointer;  { running index }
  begin if (name_length + length(area) > file_name_size) then
    begin print('File='); print_pool_str(area); print(name_of_file, ','); file_nm_size_overflow;
    end;
  name_ptr ← name_length;
  while (name_ptr > 0) do  { shift up name }
    begin name_of_file[name_ptr + length(area)] ← name_of_file[name_ptr]; decr(name_ptr);
    end;
  name_ptr ← 1; p_ptr ← str_start[area];
  while (p_ptr < str_start[area + 1]) do
    begin name_of_file[name_ptr] ← chr(str_pool[p_ptr]); incr(name_ptr); incr(p_ptr);
    end;
  name_length ← name_length + length(area);
  end;
```

62. This system-independent procedure converts upper-case characters to lower case for the specified part of *buf*. It is system independent because it uses only the internal representation for characters.

```
  define case_difference = "a" − "A"
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure lower_case(var buf : buf_type; bf_ptr, len : buf_pointer);
  var i: buf_pointer;
  begin if (len > 0) then
    for i ← bf_ptr to bf_ptr + len − 1 do
      if ((buf[i] ≥ "A") ∧ (buf[i] ≤ "Z")) then buf[i] ← buf[i] + case_difference;
  end;
```

63. This system-independent procedure is the same as the previous except that it converts lower- to upper-case letters.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure upper_case(var buf : buf_type; bf_ptr, len : buf_pointer);
  var i: buf_pointer;
  begin if (len > 0) then
    for i ← bf_ptr to bf_ptr + len − 1 do
      if ((buf[i] ≥ "a") ∧ (buf[i] ≤ "z")) then buf[i] ← buf[i] − case_difference;
  end;
```

64. The hash table. All static strings that BibTeX might have to search for, generally identifiers, are stored and retrieved by means of a fairly standard hash-table algorithm (but slightly altered here) called the method of "coalescing lists" (cf. Algorithm 6.4C in *The Art of Computer Programming*). Once a string enters the table, it is never removed. The actual sequence of characters forming a string is stored in the *str_pool* array.

The hash table consists of the four arrays *hash_next*, *hash_text*, *hash_ilk*, and *ilk_info*. The first array, *hash_next*[*p*], points to the next identifier belonging to the same coalesced list as the identifier corresponding to *p*. The second, *hash_text*[*p*], points to the *str_start* entry for *p*'s string. If position *p* of the hash table is empty, we have *hash_text*[*p*] = 0; if position *p* is either empty or the end of a coalesced hash list, we have *hash_next*[*p*] = *empty*; an auxiliary pointer variable called *hash_used* is maintained in such a way that all locations $p \geq$ *hash_used* are nonempty. The third, *hash_ilk*[*p*], tells how this string is used (as ordinary text, as a variable name, as an `.aux` file command, etc). The fourth, *ilk_info*[*p*], contains information specific to the corresponding *hash_ilk*—for *integer_ilk*s: the integer's value; for *cite_ilk*s: a pointer into *cite_list*; for *lc_cite_ilk*s: a pointer to a *cite_ilk* string; for *command_ilk*s: a constant to be used in a **case** statement; for *bst_fn_ilk*s: function-specific information; for *macro_ilk*s: a pointer to its definition string; for *control_seq_ilk*s: a constant for use in a **case** statement; for all other *ilk*s it contains no information. This *ilk*-specific information is set in other parts of the program rather than here in the hashing routine.

define *hash_base* = *empty* + 1 { lowest numbered hash-table location }
define *hash_max* = *hash_base* + *hash_size* − 1 { highest numbered hash-table location }
define *hash_is_full* ≡ (*hash_used* = *hash_base*) { test if all positions are occupied }

define *text_ilk* = 0 { a string of ordinary text }
define *integer_ilk* = 1 { an integer (possibly with a *minus_sign*) }
define *aux_command_ilk* = 2 { an `.aux`-file command }
define *aux_file_ilk* = 3 { an `.aux` file name }
define *bst_command_ilk* = 4 { a `.bst`-file command }
define *bst_file_ilk* = 5 { a `.bst` file name }
define *bib_file_ilk* = 6 { a `.bib` file name }
define *file_ext_ilk* = 7 { one of `.aux`, `.bst`, `.bib`, `.bbl`, or `.blg` }
define *file_area_ilk* = 8 { one of `texinputs:` or `texbib:` }
define *cite_ilk* = 9 { a `\citation` argument }
define *lc_cite_ilk* = 10 { a `\citation` argument converted to lower case }
define *bst_fn_ilk* = 11 { a `.bst` function name }
define *bib_command_ilk* = 12 { a `.bib`-file command }
define *macro_ilk* = 13 { a `.bst` macro or a `.bib` string }
define *control_seq_ilk* = 14 { a control sequence specifying a foreign character }
define *last_ilk* = 14 { the same number as on the line above }

⟨ Types in the outer block 22 ⟩ +≡
hash_loc = *hash_base* .. *hash_max*; { a location within the hash table }
hash_pointer = *empty* .. *hash_max*; { either *empty* or a *hash_loc* }
str_ilk = 0 .. *last_ilk*; { the legal string types }

65.

⟨ Globals in the outer block 16 ⟩ +≡
hash_next: **packed array** [*hash_loc*] **of** *hash_pointer*; { coalesced-list link }
hash_text: **packed array** [*hash_loc*] **of** *str_number*; { pointer to a string }
hash_ilk: **packed array** [*hash_loc*] **of** *str_ilk*; { the type of string }
ilk_info: **packed array** [*hash_loc*] **of** *integer*; { *ilk*-specific info }
hash_used: *hash_base* .. *hash_max* + 1; { allocation pointer for hash table }
hash_found: *boolean*; { set to *true* if it's already in the hash table }
dummy_loc: *hash_loc*; { receives *str_lookup* value whenever it's useless }

66.

⟨ Local variables for initialization 23 ⟩ +≡
k: *hash_loc*;

67. Now it's time to initialize the hash table; note that *str_start*[0] must be unused if *hash_text*[*k*] ← 0 is to have the desired effect.

⟨ Set initial values of key variables 20 ⟩ +≡
for *k* ← *hash_base* **to** *hash_max* **do**
begin *hash_next*[*k*] ← *empty*; *hash_text*[*k*] ← 0; { thus, no need to initialize *hash_ilk* or *ilk_info* }
end;
hash_used ← *hash_max* + 1; { nothing in table initially }

68. Here is the subroutine that searches the hash table for a (string, *str_ilk*) pair, where the string is of length $l \geq 0$ and appears in *buffer*[j .. ($j+l-1$)]. If it finds the pair, it returns the corresponding hash-table location and sets the global variable *hash_found* to *true*. Otherwise it sets *hash_found* to *false*, and if the parameter *insert_it* is *true*, it inserts the pair into the hash table, inserts the string into *str_pool* if not previously encountered, and returns its location. Note that two different pairs can have the same string but different *str_ilk*s, in which case the second pair encountered, if *insert_it* were *true*, would be inserted into the hash table though its string wouldn't be inserted into *str_pool* because it would already be there.

define *max_hash_value* = *hash_prime* + *hash_prime* − 2 + 127 { *h*'s maximum value }
define *do_insert* ≡ *true* { insert string if not found in hash table }
define *dont_insert* ≡ *false* { don't insert string }
define *str_found* = 40 { go here when you've found the string }
define *str_not_found* = 45 { go here when you haven't }

⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
function *str_lookup*(**var** *buf* : *buf_type*; *j*, *l* : *buf_pointer*; *ilk* : *str_ilk*; *insert_it* : *boolean*): *hash_loc*;
{ search the hash table }
label *str_found*, *str_not_found*;
var *h*: 0 .. *max_hash_value*; { hash code }
p: *hash_loc*; { index into *hash_* arrays }
k: *buf_pointer*; { index into *buf* array }
old_string: *boolean*; { set to *true* if it's an already encountered string }
str_num: *str_number*; { pointer to an already encountered string }
begin ⟨ Compute the hash code *h* 69 ⟩;
p ← *h* + *hash_base*; { start searching here; note that 0 ≤ *h* < *hash_prime* }
hash_found ← *false*; *old_string* ← *false*;
loop
begin ⟨ Process the string if we've already encountered it 70 ⟩;
if (*hash_next*[*p*] = *empty*) **then** { location *p* may or may not be empty }
begin if (¬*insert_it*) **then goto** *str_not_found*;
⟨ Insert pair into hash table and make *p* point to it 71 ⟩;
goto *str_found*;
end;
p ← *hash_next*[*p*]; { old and new locations *p* are not empty }
end;
str_not_found: *do_nothing*; { don't insert pair; function value meaningless }
str_found: *str_lookup* ← *p*;
end;

69. The value of *hash_prime* should be roughly 85% of *hash_size*, and it should be a prime number (it should also be less than $2^{14} + 2^6 = 16320$ because of WEB's simple-macro bound). The theory of hashing tells us to expect fewer than two table probes, on the average, when the search is successful.

```
⟨ Compute the hash code h 69 ⟩ ≡
  begin h ← 0;  { note that this works for zero-length strings }
  k ← j;
  while (k < j + l) do  { not a for loop in case j = l = 0 }
    begin h ← h + h + buf[k];
    while (h ≥ hash_prime) do h ← h − hash_prime;
    incr(k);
    end;
  end
```

This code is used in section 68.

70. Here we handle the case in which we've already encountered this string; note that even if we have, we'll still have to insert the pair into the hash table if *str_ilk* doesn't match.

```
⟨ Process the string if we've already encountered it 70 ⟩ ≡
  begin if (hash_text[p] > 0) then  { there's something here }
    if (str_eq_buf(hash_text[p], buf, j, l)) then  { it's the right string }
      if (hash_ilk[p] = ilk) then  { it's the right str_ilk }
        begin hash_found ← true; goto str_found;
        end
      else begin  { it's the wrong str_ilk }
        old_string ← true; str_num ← hash_text[p];
        end;
  end
```

This code is used in section 68.

71. This code inserts the pair in the appropriate unused location.

```
⟨ Insert pair into hash table and make p point to it 71 ⟩ ≡
  begin if (hash_text[p] > 0) then  { location p isn't empty }
    begin repeat if (hash_is_full) then overflow('hash␣size␣', hash_size);
      decr(hash_used);
    until (hash_text[hash_used] = 0);  { search for an empty location }
    hash_next[p] ← hash_used; p ← hash_used;
    end;  { now location p is empty }
  if (old_string) then  { it's an already encountered string }
    hash_text[p] ← str_num
  else begin  { it's a new string }
    str_room(l);  { make sure it'll fit in str_pool }
    k ← j;
    while (k < j + l) do  { not a for loop in case j = l = 0 }
      begin append_char(buf[k]); incr(k);
      end;
    hash_text[p] ← make_string;  { and make it official }
    end;
  hash_ilk[p] ← ilk;
  end
```

This code is used in section 68.

72. Now that we've defined the hash-table workings we can initialize the string pool. Unlike TEX, BIBTEX does not use a *pool_file* for string storage; instead it inserts its pre-defined strings into *str_pool*—this makes one file fewer for the BIBTEX implementor to deal with. This section initializes *str_pool*; the pre-defined strings will be inserted into it shortly; and other strings are inserted while processing the input files.

⟨ Set initial values of key variables 20 ⟩ +≡
  pool_ptr ← 0; *str_ptr* ← 1; { hash table must have *str_start*[0] unused }
  str_start[*str_ptr*] ← *pool_ptr*;

73. The longest pre-defined string determines type definitions used to insert the pre-defined strings into *str_pool*.

  define *longest_pds* = 12 { the length of '`change.case$`' }

⟨ Types in the outer block 22 ⟩ +≡
  pds_loc = 1 .. *longest_pds*; *pds_len* = 0 .. *longest_pds*; *pds_type* = **packed array** [*pds_loc*] **of** *char*;

74. The variables in this program beginning with *s_* specify the locations in *str_pool* for certain often-used strings. Those here have to do with the file system; the next section will actually insert them into *str_pool*.

⟨ Globals in the outer block 16 ⟩ +≡
s_aux_extension: *str_number*; { `.aux` }
s_log_extension: *str_number*; { `.blg` }
s_bbl_extension: *str_number*; { `.bbl` }
s_bst_extension: *str_number*; { `.bst` }
s_bib_extension: *str_number*; { `.bib` }
s_bst_area: *str_number*; { `texinputs:` }
s_bib_area: *str_number*; { `texbib:` }

75. It's time to insert some of the pre-defined strings into *str_pool* (and thus the hash table). These system-dependent strings should contain no upper-case letters, and they must all be exactly *longest_pds* characters long (even if fewer characters are actually stored). The *pre_define* routine appears shortly.

Important notes: These pre-definitions must not have any glitches or the program may bomb because the *log_file* hasn't been opened yet, and *text_ilk*s should be pre-defined later, for `.bst`-function-execution purposes.

⟨ Pre-define certain strings 75 ⟩ ≡
  pre_define(´`.aux␣␣␣␣␣␣␣␣`´, 4, *file_ext_ilk*); *s_aux_extension* ← *hash_text*[*pre_def_loc*];
  pre_define(´`.bbl␣␣␣␣␣␣␣␣`´, 4, *file_ext_ilk*); *s_bbl_extension* ← *hash_text*[*pre_def_loc*];
  pre_define(´`.blg␣␣␣␣␣␣␣␣`´, 4, *file_ext_ilk*); *s_log_extension* ← *hash_text*[*pre_def_loc*];
  pre_define(´`.bst␣␣␣␣␣␣␣␣`´, 4, *file_ext_ilk*); *s_bst_extension* ← *hash_text*[*pre_def_loc*];
  pre_define(´`.bib␣␣␣␣␣␣␣␣`´, 4, *file_ext_ilk*); *s_bib_extension* ← *hash_text*[*pre_def_loc*];
  pre_define(´`texinputs:␣␣`´, 10, *file_area_ilk*); *s_bst_area* ← *hash_text*[*pre_def_loc*];
  pre_define(´`texbib:␣␣␣␣␣`´, 7, *file_area_ilk*); *s_bib_area* ← *hash_text*[*pre_def_loc*];

See also sections 79, 334, 339·$10^T s340$.

This code is used in section 336.

76. This global variable gives the hash-table location of pre-defined strings generated by calls to *str_lookup*.

⟨ Globals in the outer block 16 ⟩ +≡
pre_def_loc: *hash_loc*;

77. This procedure initializes a pre-defined string of length at most *longest_pds*.

⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure *pre_define*(*pds* : *pds_type*; *len* : *pds_len*; *ilk* : *str_ilk*);
  var *i*: *pds_len*;
  begin for *i* ← 1 **to** *len* **do** *buffer*[*i*] ← *xord*[*pds*[*i*]];
  pre_def_loc ← *str_lookup*(*buffer*, 1, *len*, *ilk*, *do_insert*);
  end;

78. These constants all begin with *n_* and are used for the **case** statement that determines which command to execute. The variable *command_num* is set to one of these and is used to do the branching, but it must have the full *integer* range because at times it can assume an arbitrary *ilk_info* value (though it will be one of the values here when we actually use it).

  define *n_aux_bibdata* = 0 { `\bibdata` }
  define *n_aux_bibstyle* = 1 { `\bibstyle` }
  define *n_aux_citation* = 2 { `\citation` }
  define *n_aux_input* = 3 { `\@input` }
  define *n_bst_entry* = 0 { `entry` }
  define *n_bst_execute* = 1 { `execute` }
  define *n_bst_function* = 2 { `function` }
  define *n_bst_integers* = 3 { `integers` }
  define *n_bst_iterate* = 4 { `iterate` }
  define *n_bst_macro* = 5 { `macro` }
  define *n_bst_read* = 6 { `read` }
  define *n_bst_reverse* = 7 { `reverse` }
  define *n_bst_sort* = 8 { `sort` }
  define *n_bst_strings* = 9 { `strings` }
  define *n_bib_comment* = 0 { `comment` }
  define *n_bib_preamble* = 1 { `preamble` }
  define *n_bib_string* = 2 { `string` }

⟨ Globals in the outer block 16 ⟩ +≡
command_num: *integer*;

79. Now we pre-define the command strings; they must all be exactly *longest_pds* characters long.

Important note: These pre-definitions must not have any glitches or the program may bomb because the *log_file* hasn't been opened yet.

⟨ Pre-define certain strings 75 ⟩ +≡

```
pre_define('\citation␣␣␣', 9, aux_command_ilk); ilk_info[pre_def_loc] ← n_aux_citation;
pre_define('\bibdata␣␣␣␣', 8, aux_command_ilk); ilk_info[pre_def_loc] ← n_aux_bibdata;
pre_define('\bibstyle␣␣␣', 9, aux_command_ilk); ilk_info[pre_def_loc] ← n_aux_bibstyle;
pre_define('\@input␣␣␣␣␣', 7, aux_command_ilk); ilk_info[pre_def_loc] ← n_aux_input;
pre_define('entry␣␣␣␣␣␣␣', 5, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_entry;
pre_define('execute␣␣␣␣␣', 7, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_execute;
pre_define('function␣␣␣␣', 8, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_function;
pre_define('integers␣␣␣␣', 8, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_integers;
pre_define('iterate␣␣␣␣␣', 7, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_iterate;
pre_define('macro␣␣␣␣␣␣␣', 5, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_macro;
pre_define('read␣␣␣␣␣␣␣␣', 4, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_read;
pre_define('reverse␣␣␣␣␣', 7, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_reverse;
pre_define('sort␣␣␣␣␣␣␣␣', 4, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_sort;
pre_define('strings␣␣␣␣␣', 7, bst_command_ilk); ilk_info[pre_def_loc] ← n_bst_strings;
pre_define('comment␣␣␣␣␣', 7, bib_command_ilk); ilk_info[pre_def_loc] ← n_bib_comment;
pre_define('preamble␣␣␣␣', 8, bib_command_ilk); ilk_info[pre_def_loc] ← n_bib_preamble;
pre_define('string␣␣␣␣␣␣', 6, bib_command_ilk); ilk_info[pre_def_loc] ← n_bib_string;
```

80. Scanning an input line. This section describes the various *buffer* scanning routines. The two global variables *buf_ptr1* and *buf_ptr2* are used in scanning an input line. Between scans, *buf_ptr1* points to the first character of the current token and *buf_ptr2* points to that of the next. The global variable *last*, set by the function *input_ln*, marks the end of the current line; it equals 0 at the end of the current file. All the procedures and functions in this section will indicate an end-of-line when it's the end of the file.

```
define token_len ≡ (buf_ptr2 − buf_ptr1)  { of the current token }
define scan_char ≡ buffer[buf_ptr2]  { the current character }
⟨ Globals in the outer block 16 ⟩ +≡
buf_ptr1: buf_pointer;  { points to the first position of the current token }
buf_ptr2: buf_pointer;  { used to find the end of the current token }
```

81. These macros send the current token, in *buffer*[*buf_ptr1*] to *buffer*[*buf_ptr2* − 1], to an output file.

```
define print_token ≡ print_a_token  { making this a procedure saves a little space }
define trace_pr_token ≡
        begin out_token(log_file);
        end
```

82. And here are the associated procedures. Note: The *term_out* file is system dependent.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure out_token(var f : alpha_file);
  var i: buf_pointer;
  begin i ← buf_ptr1;
  while (i < buf_ptr2) do
    begin write(f, xchr[buffer[i]]); incr(i);
    end;
  end;
procedure print_a_token;
  begin out_token(term_out); out_token(log_file);
  end;
```

83. This function scans the *buffer* for the next token, starting at the global variable *buf_ptr2* and ending just before either the single specified stop-character or the end of the current line, whichever comes first, respectively returning *true* or *false*; afterward, *scan_char* is the first character following this token.

```
⟨ Procedures and functions for input scanning 83 ⟩ ≡
function scan1(char1 : ASCII_code): boolean;
  begin buf_ptr1 ← buf_ptr2;  { scan until end-of-line or the specified character }
  while ((scan_char ≠ char1) ∧ (buf_ptr2 < last)) do incr(buf_ptr2);
  if (buf_ptr2 < last) then scan1 ← true
  else scan1 ← false;
  end;
```

See also sections 84, 85, 86, 87, 88, 90, 92, 93, 94, 152, 183, 184, 185, 186, 187, 228, 248·10^T s249.

This code is used in section 12.

84. This function is the same but stops at *white_space* characters as well.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan1_white(char1 : ASCII_code): boolean;
  begin buf_ptr1 ← buf_ptr2; { scan until end-of-line, the specified character, or white_space }
  while ((lex_class[scan_char] ≠ white_space) ∧ (scan_char ≠ char1) ∧ (buf_ptr2 < last)) do
    incr(buf_ptr2);
  if (buf_ptr2 < last) then scan1_white ← true
  else scan1_white ← false;
  end;
```

85. This function is similar to *scan1*, but stops at either of two stop-characters as well as the end of the current line.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan2(char1, char2 : ASCII_code): boolean;
  begin buf_ptr1 ← buf_ptr2; { scan until end-of-line or the specified characters }
  while ((scan_char ≠ char1) ∧ (scan_char ≠ char2) ∧ (buf_ptr2 < last)) do incr(buf_ptr2);
  if (buf_ptr2 < last) then scan2 ← true
  else scan2 ← false;
  end;
```

86. This function is the same but stops at *white_space* characters as well.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan2_white(char1, char2 : ASCII_code): boolean;
  begin buf_ptr1 ← buf_ptr2; { scan until end-of-line, the specified characters, or white_space }
  while ((scan_char ≠ char1)∧(scan_char ≠ char2)∧(lex_class[scan_char] ≠ white_space)∧(buf_ptr2 < last))
      do incr(buf_ptr2);
  if (buf_ptr2 < last) then scan2_white ← true
  else scan2_white ← false;
  end;
```

87. This function is similar to *scan2*, but stops at either of three stop-characters as well as the end of the current line.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan3(char1, char2, char3 : ASCII_code): boolean;
  begin buf_ptr1 ← buf_ptr2; { scan until end-of-line or the specified characters }
  while ((scan_char ≠ char1) ∧ (scan_char ≠ char2) ∧ (scan_char ≠ char3) ∧ (buf_ptr2 < last)) do
    incr(buf_ptr2);
  if (buf_ptr2 < last) then scan3 ← true
  else scan3 ← false;
  end;
```

88. This function scans for letters, stopping at the first nonletter; it returns *true* if there is at least one letter.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan_alpha: boolean;
  begin buf_ptr1 ← buf_ptr2; { scan until end-of-line or a nonletter }
  while ((lex_class[scan_char] = alpha) ∧ (buf_ptr2 < last)) do incr(buf_ptr2);
  if (token_len = 0) then scan_alpha ← false
  else scan_alpha ← true;
  end;
```

89. These are the possible values for *scan_result*; they're set by the *scan_identifier* procedure and are described in the next section.

```
define id_null = 0
define specified_char_adjacent = 1
define other_char_adjacent = 2
define white_adjacent = 3
```

```
⟨ Globals in the outer block 16 ⟩ +≡
scan_result: id_null .. white_adjacent;
```

90. This procedure scans for an identifier, stopping at the first *illegal_id_char*, or stopping at the first character if it's *numeric*. It sets the global variable *scan_result* to *id_null* if the identifier is null, else to *white_adjacent* if it ended at a *white_space* character or an end-of-line, else to *specified_char_adjacent* if it ended at one of *char1* or *char2* or *char3*, else to *other_char_adjacent* if it ended at a nonspecified, non*white_space illegal_id_char*. By convention, when some calling code really wants just one or two "specified" characters, it merely repeats one of the characters.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
procedure scan_identifier(char1, char2, char3 : ASCII_code);
  begin buf_ptr1 ← buf_ptr2;
  if (lex_class[scan_char] ≠ numeric) then { scan until end-of-line or an illegal_id_char }
    while ((id_class[scan_char] = legal_id_char) ∧ (buf_ptr2 < last)) do incr(buf_ptr2);
  if (token_len = 0) then scan_result ← id_null
  else if ((lex_class[scan_char] = white_space) ∨ (buf_ptr2 = last)) then scan_result ← white_adjacent
    else if ((scan_char = char1) ∨ (scan_char = char2) ∨ (scan_char = char3)) then
        scan_result ← specified_char_adjacent
      else scan_result ← other_char_adjacent;
  end;
```

91. The next two procedures scan for an integer, setting the global variable *token_value* to the corresponding integer.

```
define char_value ≡ (scan_char − "0")  { the value of the digit being scanned }
```

```
⟨ Globals in the outer block 16 ⟩ +≡
token_value: integer;  { the numeric value of the current token }
```

92. This function scans for a nonnegative integer, stopping at the first nondigit; it sets the value of *token_value* accordingly. It returns *true* if the token was a legal nonnegative integer (i.e., consisted of one or more digits).

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan_nonneg_integer: boolean;
  begin buf_ptr1 ← buf_ptr2; token_value ← 0;  { scan until end-of-line or a nondigit }
  while ((lex_class[scan_char] = numeric) ∧ (buf_ptr2 < last)) do
    begin token_value ← token_value ∗ 10 + char_value; incr(buf_ptr2);
    end;
  if (token_len = 0) then  { there were no digits }
    scan_nonneg_integer ← false
  else scan_nonneg_integer ← true;
  end;
```

93. This procedure scans for an integer, stopping at the first nondigit; it sets the value of *token_value* accordingly. It returns *true* if the token was a legal integer (i.e., consisted of an optional *minus_sign* followed by one or more digits).

```
define negative ≡ (sign_length = 1)  { if this integer is negative }
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan_integer: boolean;
  var sign_length: 0 .. 1;  { 1 if there's a minus_sign, 0 if not }
  begin buf_ptr1 ← buf_ptr2;
  if (scan_char = minus_sign) then  { it's a negative number }
    begin sign_length ← 1; incr(buf_ptr2);  { skip over the minus_sign }
    end
  else sign_length ← 0;
  token_value ← 0;  { scan until end-of-line or a nondigit }
  while ((lex_class[scan_char] = numeric) ∧ (buf_ptr2 < last)) do
    begin token_value ← token_value * 10 + char_value; incr(buf_ptr2);
    end;
  if (negative) then token_value ← −token_value;
  if (token_len = sign_length) then  { there were no digits }
    scan_integer ← false
  else scan_integer ← true;
  end;
```

94. This function scans over *white_space* characters, stopping either at the first nonwhite character or the end of the line, respectively returning *true* or *false*.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan_white_space: boolean;
  begin  { scan until end-of-line or a nonwhite }
  while ((lex_class[scan_char] = white_space) ∧ (buf_ptr2 < last)) do incr(buf_ptr2);
  if (buf_ptr2 < last) then scan_white_space ← true
  else scan_white_space ← false;
  end;
```

95. The *print_bad_input_line* procedure prints the current input line, splitting it at the character being scanned: It prints *buffer*[0], *buffer*[1], ..., *buffer*[*buf_ptr2* − 1] on one line and *buffer*[*buf_ptr2*], ..., *buffer*[*last* − 1] on the next (and both lines start with a colon between two *space*s). Each *white_space* character is printed as a *space*.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
```
procedure print_bad_input_line;
  var bf_ptr: buf_pointer;
  begin print(´␣:␣´); bf_ptr ← 0;
  while (bf_ptr < buf_ptr2) do
    begin if (lex_class[buffer[bf_ptr]] = white_space) then print(xchr[space])
    else print(xchr[buffer[bf_ptr]]);
    incr(bf_ptr);
    end;
  print_newline; print(´␣:␣´); bf_ptr ← 0;
  while (bf_ptr < buf_ptr2) do
    begin print(xchr[space]); incr(bf_ptr);
    end;
  bf_ptr ← buf_ptr2;
  while (bf_ptr < last) do
    begin if (lex_class[buffer[bf_ptr]] = white_space) then print(xchr[space])
    else print(xchr[buffer[bf_ptr]]);
    incr(bf_ptr);
    end;
  print_newline;
  bf_ptr ← 0;
  while ((bf_ptr < buf_ptr2) ∧ (lex_class[buffer[bf_ptr]] = white_space)) do incr(bf_ptr);
  if (bf_ptr = buf_ptr2) then print_ln(´(Error␣may␣have␣been␣on␣previous␣line)´);
  mark_error;
  end;
```

96. This little procedure exists because it's used by at least two other procedures and thus saves some space.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
```
procedure print_skipping_whatever_remains;
  begin print(´I´´m␣skipping␣whatever␣remains␣of␣this␣´);
  end;
```

97. Getting the top-level auxiliary file name. These modules read the name of the top-level `.aux` file. Some systems will try to find this on the command line; if it's not there it will come from the user's terminal. In either case, the name goes into the *char* array *name_of_file*, and the files relevant to this name are opened.

define *aux_found* = 41 { go here when the `.aux` name is legit }
define *aux_not_found* = 46 { go here when it's not }

⟨ Globals in the outer block 16 ⟩ +≡
aux_name_length: 0 .. *file_name_size* + 1; { `.aux` name sans extension }

98. I mean, this is truly disgraceful. A user has to type something in to the terminal just once during the entire run. And it's not some complicated string where you have to get every last punctuation mark just right, and it's not some fancy list where you get nervous because if you forget one item you have to type the whole thing again; it's just a simple, ordinary, file name. Now you'd think a five-year-old could do it; you'd think it's so simple a user should be able to do it in his sleep. But nooooooooooo. He had to sit there droning on and on about who knows what until he exceeded the bounds of common sense, and he probably didn't even realize it. Just pitiful. What's this world coming to? We should probably just delete all his files and be done with him. Note: The *term_out* file is system dependent.

define *sam_you_made_the_file_name_too_long* ≡
begin *sam_too_long_file_name_print*; **goto** *aux_not_found*;
end

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *sam_too_long_file_name_print*;
begin *write*(*term_out*, ´File␣name␣`´); *name_ptr* ← 1;
while (*name_ptr* ≤ *aux_name_length*) **do**
begin *write*(*term_out*, *name_of_file*[*name_ptr*]); *incr*(*name_ptr*);
end;
write_ln(*term_out*, ´´´␣is␣too␣long´);
end;

99. We've abused the user enough for one section; suffice it to say here that most of what we said last module still applies. Note: The *term_out* file is system dependent.

define *sam_you_made_the_file_name_wrong* ≡
begin *sam_wrong_file_name_print*; **goto** *aux_not_found*;
end

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *sam_wrong_file_name_print*;
begin *write*(*term_out*, ´I␣couldn´´t␣open␣file␣name␣`´); *name_ptr* ← 1;
while (*name_ptr* ≤ *name_length*) **do**
begin *write*(*term_out*, *name_of_file*[*name_ptr*]); *incr*(*name_ptr*);
end;
write_ln(*term_out*, ´´´´);
end;

100. This procedure consists of a loop that reads and processes a (nonnull) `.aux` file name. It's this module and the next two that must be changed on those systems using command-line arguments. Note: The *term_out* and *term_in* files are system dependent.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ ≡
procedure get_the_top_level_aux_file_name;
  label aux_found, aux_not_found;
  var ⟨ Variables for possible command-line processing 101 ⟩
  begin check_cmnd_line ← false;  { many systems will change this }
  loop
    begin if (check_cmnd_line) then ⟨ Process a possible command line 102 ⟩
    else begin write(term_out, 'Please␣type␣input␣file␣name␣(no␣extension)--');
      if (eoln(term_in)) then  { so the first read works }
        read_ln(term_in);
      aux_name_length ← 0;
      while (¬eoln(term_in)) do
        begin if (aux_name_length = file_name_size) then
          begin while (¬eoln(term_in)) do  { discard the rest of the line }
            get(term_in);
          sam_you_made_the_file_name_too_long;
          end;
        incr(aux_name_length); name_of_file[aux_name_length] ← term_in↑; get(term_in);
        end;
      end;
    ⟨ Handle this .aux name 103 ⟩;
  aux_not_found: check_cmnd_line ← false;
    end;
aux_found:  { now we're ready to read the .aux file }
  end;
```

See also sections 120, 126, 132, 139, 142, 143, 145, 170, 177, 178, 180, 201, 203, 205, 210, 211, 212, 214, 215·$10^T s$217.

This code is used in section 12.

101. The switch *check_cmnd_line* tells us whether we're to check for a possible command-line argument.

```
⟨ Variables for possible command-line processing 101 ⟩ ≡
check_cmnd_line: boolean;  { true if we're to check the command line }
```

This code is used in section 100.

102. Here's where we do the real command-line work. Those systems needing more than a single module to handle the task should add the extras to the "System-dependent changes" section.

```
⟨ Process a possible command line 102 ⟩ ≡
  begin do_nothing;  { the "default system" doesn't use the command line }
  end
```

This code is used in section 100.

103. Here we orchestrate this `.aux` name's handling: we add the various extensions, try to open the files with the resulting name, and store the name strings we'll need later.

```
⟨ Handle this .aux name 103 ⟩ ≡
  begin if ((aux_name_length + length(s_aux_extension) > file_name_size) ∨
        (aux_name_length + length(s_log_extension) > file_name_size) ∨
        (aux_name_length + length(s_bbl_extension) > file_name_size)) then
    sam_you_made_the_file_name_too_long;
  ⟨ Add extensions and open files 106 ⟩;
  ⟨ Put this name into the hash table 107 ⟩;
  goto aux_found;
  end
```

This code is used in section 100.

104. Here we set up definitions and declarations for files opened in this section. Each element in *aux_list* (except for *aux_list*[*aux_stack_size*], which is always unused) is a pointer to the appropriate *str_pool* string representing the `.aux` file name. The array *aux_file* contains the corresponding PASCAL **file** variables.

```
  define cur_aux_str ≡ aux_list[aux_ptr]  { shorthand for the current .aux file }
  define cur_aux_file ≡ aux_file[aux_ptr]  { shorthand for the current aux_file }
  define cur_aux_line ≡ aux_ln_stack[aux_ptr]  { line number of current .aux file }

⟨ Globals in the outer block 16 ⟩ +≡
aux_file: array [aux_number] of alpha_file;  { open .aux file variables }
aux_list: array [aux_number] of str_number;  { the open .aux file list }
aux_ptr: aux_number;  { points to the currently open .aux file }
aux_ln_stack: array [aux_number] of integer;  { open .aux line numbers }
top_lev_str: str_number;  { the top-level .aux file's name }

log_file: alpha_file;  { the file variable for the .blg file }
bbl_file: alpha_file;  { the file variable for the .bbl file }
```

105. Where *aux_number* is the obvious.

```
⟨ Types in the outer block 22 ⟩ +≡
  aux_number = 0 .. aux_stack_size;  { gives the aux_list range }
```

106. We must make sure the (top-level) `.aux`, `.blg`, and `.bbl` files can be opened.

```
⟨ Add extensions and open files 106 ⟩ ≡
  begin name_length ← aux_name_length;  { set to last used position }
  add_extension(s_aux_extension);  { this also sets name_length }
  aux_ptr ← 0;  { initialize the .aux file stack }
  if (¬a_open_in(cur_aux_file)) then sam_you_made_the_file_name_wrong;
  name_length ← aux_name_length; add_extension(s_log_extension);  { this also sets name_length }
  if (¬a_open_out(log_file)) then sam_you_made_the_file_name_wrong;
  name_length ← aux_name_length; add_extension(s_bbl_extension);  { this also sets name_length }
  if (¬a_open_out(bbl_file)) then sam_you_made_the_file_name_wrong;
  end
```

This code is used in section 103.

107. This code puts the `.aux` file name, both with and without the extension, into the hash table, and it initializes *aux_list*. Note that all previous top-level `.aux`-file stuff must have been successful.

```
⟨ Put this name into the hash table 107 ⟩ ≡
  begin name_length ← aux_name_length; add_extension(s_aux_extension); { this also sets name_length }
  name_ptr ← 1;
  while (name_ptr ≤ name_length) do
    begin buffer[name_ptr] ← xord[name_of_file[name_ptr]]; incr(name_ptr);
    end;
  top_lev_str ← hash_text[str_lookup(buffer, 1, aux_name_length, text_ilk, do_insert)];
  cur_aux_str ← hash_text[str_lookup(buffer, 1, name_length, aux_file_ilk, do_insert)];
      { note that this has initialized aux_list }
  if (hash_found) then
    begin trace print_aux_name;
    ecart
    confusion('Already␣encountered␣auxiliary␣file');
    end;
  cur_aux_line ← 0; { this finishes initializing the top-level .aux file }
  end
```

This code is used in section 103.

108. Print the name of the current `.aux` file, followed by a *newline*.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure print_aux_name;
  begin print_pool_str(cur_aux_str); print_newline;
  end;
```

109. Reading the auxiliary file(s). Now it's time to read the `.aux` file. The only commands we handle are `\citation` (there can be arbitrarily many, each having arbitrarily many arguments), `\bibdata` (there can be just one, but it can have arbitrarily many arguments), `\bibstyle` (there can be just one, and it can have just one argument), and `\@input` (there can be arbitrarily many, each with one argument, and they can be nested to a depth of *aux_stack_size*). Each of these commands is assumed to be on just a single line. The rest of the `.aux` file is ignored.

define *aux_done* = 31 { go here when finished with the `.aux` files }

⟨ Labels in the outer block 109 ⟩ ≡
, *aux_done*

See also section 146.

This code is used in section 10.

110. We keep reading and processing input lines until none left. This is part of the main program; hence, because of the *aux_done* label, there's no conventional **begin** - **end** pair surrounding the entire module.

⟨ Read the `.aux` file 110 ⟩ ≡
print(´`The␣top-level␣auxiliary␣file:␣`´); *print_aux_name*;
loop
begin { *pop_the_aux_stack* will exit the loop }
incr(*cur_aux_line*);
if (¬*input_ln*(*cur_aux_file*)) **then** { end of current `.aux` file }
pop_the_aux_stack
else *get_aux_command_and_process*;
end;
trace *trace_pr_ln*(´`Finished␣reading␣the␣auxiliary␣file(s)`´);
ecart
aux_done: *last_check_for_aux_errors*;

This code is used in section 10.

111. When we find a bug, we print a message and flush the rest of the line. This macro must be called from within a procedure that has an *exit* label.

define *aux_err_return* ≡
begin *aux_err_print*; **return**; { flush this input line }
end
define *aux_err*(#) ≡
begin *print*(#); *aux_err_return*;
end

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *aux_err_print*;
begin *print*(´`---line␣`´, *cur_aux_line* : 0, ´`␣of␣file␣`´); *print_aux_name*;
print_bad_input_line; { this call does the *mark_error* }
print_skipping_whatever_remains; *print_ln*(´`command`´)
end;

112. Here are a bunch of macros whose print statements are used at least twice. Thus we save space by making the statements procedures. This macro complains when there's a repeated command that's to be used just once.

```
define aux_err_illegal_another(#) ≡
        begin aux_err_illegal_another_print(#); aux_err_return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure aux_err_illegal_another_print(cmd_num : integer);
  begin print('Illegal, another \bib');
  case (cmd_num) of
  n_aux_bibdata: print('data');
  n_aux_bibstyle: print('style');
  othercases confusion('Illegal auxiliary-file command')
  endcases; print(' command');
  end;
```

113. This one complains when a command is missing its *right_brace*.

```
define aux_err_no_right_brace ≡
        begin aux_err_no_right_brace_print; aux_err_return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure aux_err_no_right_brace_print;
  begin print('No "', xchr[right_brace], '"');
  end;
```

114. This one complains when a command has stuff after its *right_brace*.

```
define aux_err_stuff_after_right_brace ≡
        begin aux_err_stuff_after_right_brace_print; aux_err_return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure aux_err_stuff_after_right_brace_print;
  begin print('Stuff after "', xchr[right_brace], '"');
  end;
```

115. And this one complains when a command has *white_space* in its argument.

```
define aux_err_white_space_in_argument ≡
        begin aux_err_white_space_in_argument_print; aux_err_return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure aux_err_white_space_in_argument_print;
  begin print('White space in argument');
  end;
```

116. We're not at the end of an `.aux` file, so we see if the current line might be a command of interest. A command of interest will be a line without blanks, consisting of a command name, a *left_brace*, one or more arguments separated by commas, and a *right_brace*.

```
⟨ Scan for and process an .aux command 116 ⟩ ≡
procedure get_aux_command_and_process;
  label exit;
  begin buf_ptr2 ← 0; { mark the beginning of the next token }
  if (¬scan1(left_brace)) then { no left_brace—flush line }
    return;
  command_num ← ilk_info[str_lookup(buffer, buf_ptr1, token_len, aux_command_ilk, dont_insert)];
  if (hash_found) then
    case (command_num) of
    n_aux_bibdata: aux_bib_data_command;
    n_aux_bibstyle: aux_bib_style_command;
    n_aux_citation: aux_citation_command;
    n_aux_input: aux_input_command;
    othercases confusion('Unknown␣auxiliary-file␣command')
    endcases;
exit: end;
```

This code is used in section 143.

117. Here we introduce some variables for processing a `\bibdata` command. Each element in *bib_list* (except for *bib_list*[*max_bib_files*], which is always unused) is a pointer to the appropriate *str_pool* string representing the `.bib` file name. The array *bib_file* contains the corresponding PASCAL **file** variables.

```
  define cur_bib_str ≡ bib_list[bib_ptr] { shorthand for current .bib file }
  define cur_bib_file ≡ bib_file[bib_ptr] { shorthand for current bib_file }

⟨ Globals in the outer block 16 ⟩ +≡
bib_list: array [bib_number] of str_number; { the .bib file list }
bib_ptr: bib_number; { pointer for the current .bib file }
num_bib_files: bib_number; { the total number of .bib files }
bib_seen: boolean; { true if we've already seen a \bibdata command }
bib_file: array [bib_number] of alpha_file; { corresponding file variables }
```

118. Where *bib_number* is the obvious.

```
⟨ Types in the outer block 22 ⟩ +≡
  bib_number = 0 .. max_bib_files; { gives the bib_list range }
```

119.

```
⟨ Set initial values of key variables 20 ⟩ +≡
  bib_ptr ← 0; { this makes bib_list empty }
  bib_seen ← false; { we haven't seen a \bibdata command yet }
```

120. A `\bibdata` command will have its arguments between braces and separated by commas. There must be exactly one such command in the `.aux` file(s). All upper-case letters are converted to lower case.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure aux_bib_data_command;
  label exit;
  begin if (bib_seen) then aux_err_illegal_another(n_aux_bibdata);
  bib_seen ← true; { now we've seen a \bibdata command }
  while (scan_char ≠ right_brace) do
    begin incr(buf_ptr2); { skip over the previous stop-character }
    if (¬scan2_white(right_brace, comma)) then aux_err_no_right_brace;
    if (lex_class[scan_char] = white_space) then aux_err_white_space_in_argument;
    if ((last > buf_ptr2 + 1) ∧ (scan_char = right_brace)) then aux_err_stuff_after_right_brace;
    ⟨ Open a .bib file 123 ⟩;
    end;
exit: end;
```

121. Here's a procedure we'll need shortly. It prints the name of the current `.bib` file, followed by a *newline*.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure print_bib_name;
  begin print_pool_str(cur_bib_str); print_pool_str(s_bib_extension); print_newline;
  end;
```

122. This macro is similar to *aux_err* but it complains specifically about opening a file for a `\bibdata` command.

```
  define open_bibdata_aux_err(#) ≡
          begin print(#); print_bib_name; aux_err_return; { this does the mark_error }
          end
```

123. Now we add the just-found argument to *bib_list* if it hasn't already been encountered as a `\bibdata` argument and if, after appending the *s_bib_extension* string, the resulting file name can be opened.

```
⟨ Open a .bib file 123 ⟩ ≡
  begin if (bib_ptr = max_bib_files) then overflow('number␣of␣database␣files␣', max_bib_files);
  cur_bib_str ← hash_text[str_lookup(buffer, buf_ptr1, token_len, bib_file_ilk, do_insert)];
  if (hash_found) then { already encountered this as a \bibdata argument }
    open_bibdata_aux_err('This␣database␣file␣appears␣more␣than␣once:␣');
  start_name(cur_bib_str); add_extension(s_bib_extension);
  if (¬a_open_in(cur_bib_file)) then
    begin add_area(s_bib_area);
    if (¬a_open_in(cur_bib_file)) then open_bibdata_aux_err('I␣couldn''t␣open␣database␣file␣');
    end;
  trace trace_pr_pool_str(cur_bib_str); trace_pr_pool_str(s_bib_extension);
  trace_pr_ln('␣is␣a␣bibdata␣file');
  ecart
  incr(bib_ptr);
  end
```

This code is used in section 120.

124. Here we introduce some variables for processing a `\bibstyle` command.

⟨ Globals in the outer block 16 ⟩ +≡
bst_seen: *boolean*; { *true* if we've already seen a `\bibstyle` command }
bst_str: *str_number*; { the string number for the `.bst` file }
bst_file: *alpha_file*; { the corresponding **file** variable }

125. And we initialize.

⟨ Set initial values of key variables 20 ⟩ +≡
bst_str ← 0; { mark *bst_str* as unused }
bst_seen ← *false*; { we haven't seen a `\bibstyle` command yet }

126. A `\bibstyle` command will have exactly one argument, and it will be between braces. There must be exactly one such command in the `.aux` file(s). All upper-case letters are converted to lower case.

⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure *aux_bib_style_command*;
label *exit*;
begin if (*bst_seen*) **then** *aux_err_illegal_another*(*n_aux_bibstyle*);
bst_seen ← *true*; { now we've seen a `\bibstyle` command }
incr(*buf_ptr2*); { skip over the *left_brace* }
if (¬*scan1_white*(*right_brace*)) **then** *aux_err_no_right_brace*;
if (*lex_class*[*scan_char*] = *white_space*) **then** *aux_err_white_space_in_argument*;
if (*last* > *buf_ptr2* + 1) **then** *aux_err_stuff_after_right_brace*;
⟨ Open the `.bst` file 127 ⟩;
exit: **end**;

127. Now we open the file whose name is the just-found argument appended with the *s_bst_extension* string, if possible.

⟨ Open the `.bst` file 127 ⟩ ≡
begin *bst_str* ← *hash_text*[*str_lookup*(*buffer*, *buf_ptr1*, *token_len*, *bst_file_ilk*, *do_insert*)];
if (*hash_found*) **then**
begin trace *print_bst_name*;
ecart
confusion(´`Already␣encountered␣style␣file`´);
end;
start_name(*bst_str*); *add_extension*(*s_bst_extension*);
if (¬*a_open_in*(*bst_file*)) **then**
begin *add_area*(*s_bst_area*);
if (¬*a_open_in*(*bst_file*)) **then**
begin *print*(´`I␣couldn´´t␣open␣style␣file␣`´); *print_bst_name*;
bst_str ← 0; { mark as unused again }
aux_err_return;
end;
end;
print(´`The␣style␣file:␣`´); *print_bst_name*;
end

This code is used in section 126.

128. Print the name of the `.bst` file, followed by a *newline*.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *print_bst_name*;
  begin *print_pool_str*(*bst_str*); *print_pool_str*(*s_bst_extension*); *print_newline*;
  end;

129. Here we introduce some variables for processing a `\citation` command. Each element in *cite_list* (except for *cite_list*[*max_cites*], which is always unused) is a pointer to the appropriate *str_pool* string. The cite-key list is kept in order of occurrence with duplicates removed.

  define *cur_cite_str* ≡ *cite_list*[*cite_ptr*] { shorthand for the current cite key }

⟨ Globals in the outer block 16 ⟩ +≡
cite_list: **packed array** [*cite_number*] **of** *str_number*; { the cite-key list }
cite_ptr: *cite_number*; { pointer for the current cite key }
entry_cite_ptr: *cite_number*; { cite pointer for the current entry }
num_cites: *cite_number*; { the total number of distinct cite keys }
old_num_cites: *cite_number*; { set to a previous *num_cites* value }
citation_seen: *boolean*; { *true* if we've seen a `\citation` command }
cite_loc: *hash_loc*; { the hash-table location of a cite key }
lc_cite_loc: *hash_loc*; { and of its lower-case equivalent }
lc_xcite_loc: *hash_loc*; { a second *lc_cite_loc* variable }
cite_found: *boolean*; { *true* if we've already seen this cite key }
all_entries: *boolean*; { *true* if we're to use the entire database }
all_marker: *cite_number*; { we put the other entries in *cite_list* here }

130. Where *cite_number* is the obvious.

⟨ Types in the outer block 22 ⟩ +≡
  cite_number = 0 .. *max_cites*; { gives the *cite_list* range }

131.

⟨ Set initial values of key variables 20 ⟩ +≡
  cite_ptr ← 0; { this makes *cite_list* empty }
  citation_seen ← *false*; { we haven't seen a `\citation` command yet }
  all_entries ← *false*; { by default, use just the entries explicitly named }

132. A `\citation` command will have its arguments between braces and separated by commas. Upper/lower cases are considered to be different for `\citation` arguments, which is the same as the rest of LaTeX but different from the rest of BibTeX. A cite key needn't exactly case-match its corresponding database key to work, although two cite keys that are case-mismatched will produce an error message. (A *case mismatch* is a mismatch, but only because of a case difference.)

A `\citation` command having `*` as an argument indicates that the entire database will be included (almost as if a `\nocite` command that listed every cite key in the database, in order, had been given at the corresponding spot in the `.tex` file).

```
define next_cite = 23  { read the next argument }
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure aux_citation_command;
  label next_cite, exit;
  begin citation_seen ← true;  { now we've seen a \citation command }
  while (scan_char ≠ right_brace) do
    begin incr(buf_ptr2);  { skip over the previous stop-character }
    if (¬scan2_white(right_brace, comma)) then aux_err_no_right_brace;
    if (lex_class[scan_char] = white_space) then aux_err_white_space_in_argument;
    if ((last > buf_ptr2 + 1) ∧ (scan_char = right_brace)) then aux_err_stuff_after_right_brace;
    ⟨ Check the cite key 133 ⟩;
  next_cite: end;
exit: end;
```

133. We must check if (the lower-case version of) this cite key has been previously encountered, and proceed accordingly. The alias kludge helps make the stack space not overflow on some machines.

```
define ex_buf1 ≡ ex_buf  { an alias, used only in this module }
⟨ Check the cite key 133 ⟩ ≡
  begin trace trace_pr_token; trace_pr('␣cite␣key␣encountered');
  ecart
  ⟨ Check for entire database inclusion (and thus skip this cite key) 134 ⟩;
  tmp_ptr ← buf_ptr1;
  while (tmp_ptr < buf_ptr2) do
    begin ex_buf1[tmp_ptr] ← buffer[tmp_ptr]; incr(tmp_ptr);
    end;
  lower_case(ex_buf1, buf_ptr1, token_len);  { convert to 'canonical' form }
  lc_cite_loc ← str_lookup(ex_buf1, buf_ptr1, token_len, lc_cite_ilk, do_insert);
  if (hash_found) then  { already encountered this as a \citation argument }
    ⟨ Cite seen, don't add a cite key 135 ⟩
  else ⟨ Cite unseen, add a cite key 136 ⟩;  { it's a new cite key—add it to cite_list }
  end
```

This code is used in section 132.

134. Here we check for a `\citation` command having `*` as an argument, indicating that the entire database will be included.

⟨ Check for entire database inclusion (and thus skip this cite key) 134 ⟩ ≡
```
begin if (token_len = 1) then
  if (buffer[buf_ptr1] = star) then
    begin trace trace_pr_ln('---entire␣database␣to␣be␣included');
    ecart
    if (all_entries) then
      begin print_ln('Multiple␣inclusions␣of␣entire␣database'); aux_err_return;
      end
    else begin all_entries ← true; all_marker ← cite_ptr; goto next_cite;
      end;
    end;
end
```
This code is used in section 133.

135. We've previously encountered the lower-case version, so we check that the actual version exactly matches the actual version of the previously-encountered cite key(s).

⟨ Cite seen, don't add a cite key 135 ⟩ ≡
```
begin trace trace_pr_ln('␣previously');
ecart
dummy_loc ← str_lookup(buffer, buf_ptr1, token_len, cite_ilk, dont_insert);
if (¬hash_found) then { case mismatch error }
  begin print('Case␣mismatch␣error␣between␣cite␣keys␣'); print_token; print('␣and␣');
  print_pool_str(cite_list[ilk_info[ilk_info[lc_cite_loc]]]); print_newline; aux_err_return;
  end;
end
```
This code is used in section 133.

136. Now we add the just-found argument to *cite_list* if there isn't anything funny happening.

⟨ Cite unseen, add a cite key 136 ⟩ ≡
```
begin trace trace_pr_newline;
ecart
cite_loc ← str_lookup(buffer, buf_ptr1, token_len, cite_ilk, do_insert);
if (hash_found) then hash_cite_confusion;
check_cite_overflow(cite_ptr); cur_cite_str ← hash_text[cite_loc]; ilk_info[cite_loc] ← cite_ptr;
ilk_info[lc_cite_loc] ← cite_loc; incr(cite_ptr);
end
```
This code is used in section 133.

137. Here's a serious complaint (that is, a bug) concerning hash problems. This is the first of several similar bug-procedures that exist only because they save space.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
```
procedure hash_cite_confusion;
  begin confusion('Cite␣hash␣error');
  end;
```

138. Complain if somebody's got a cite fetish. This procedure is called when were about to add another cite key to *cite_list*. It assumes that *cite_loc* gives the potential cite key's hash table location.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure check_cite_overflow(last_cite : cite_number);
  begin if (last_cite = max_cites) then
    begin print_pool_str(hash_text[cite_loc]); print_ln(´␣is␣the␣key:´);
    overflow(´number␣of␣cite␣keys␣´, max_cites);
    end;
  end;
```

139. An `\@input` command will have exactly one argument, it will be between braces, and it must have the *s_aux_extension*. All upper-case letters are converted to lower case.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure aux_input_command;
  label exit;
  var aux_extension_ok: boolean; { to check for a correct file extension }
  begin incr(buf_ptr2); { skip over the left_brace }
  if (¬scan1_white(right_brace)) then aux_err_no_right_brace;
  if (lex_class[scan_char] = white_space) then aux_err_white_space_in_argument;
  if (last > buf_ptr2 + 1) then aux_err_stuff_after_right_brace;
  ⟨ Push the .aux stack 140 ⟩;
exit: end;
```

140. We must check that this potential `.aux` file won't overflow the stack, that it has the correct extension, that we haven't encountered it before (to prevent, among other things, an infinite loop).

```
⟨ Push the .aux stack 140 ⟩ ≡
  begin incr(aux_ptr);
  if (aux_ptr = aux_stack_size) then
    begin print_token; print(´:␣´); overflow(´auxiliary␣file␣depth␣´, aux_stack_size);
    end;
  aux_extension_ok ← true;
  if (token_len < length(s_aux_extension)) then
    aux_extension_ok ← false { else str_eq_buf might bomb the program }
  else if (¬str_eq_buf(s_aux_extension, buffer, buf_ptr2 − length(s_aux_extension), length(s_aux_extension)))
          then aux_extension_ok ← false;
  if (¬aux_extension_ok) then
    begin print_token; print(´␣has␣a␣wrong␣extension´); decr(aux_ptr); aux_err_return;
    end;
  cur_aux_str ← hash_text[str_lookup(buffer, buf_ptr1, token_len, aux_file_ilk, do_insert)];
  if (hash_found) then
    begin print(´Already␣encountered␣file␣´); print_aux_name; decr(aux_ptr); aux_err_return;
    end;
  ⟨ Open this .aux file 141 ⟩;
  end
```

This code is used in section 139.

141. We check that this `.aux` file can actually be opened, and then open it.

```
⟨ Open this .aux file 141 ⟩ ≡
  begin start_name(cur_aux_str); { extension already there for .aux files }
  name_ptr ← name_length + 1;
  while (name_ptr ≤ file_name_size) do { pad with blanks }
    begin name_of_file[name_ptr] ← ´␣´; incr(name_ptr);
    end;
  if (¬a_open_in(cur_aux_file)) then
    begin print(´I␣couldn´´t␣open␣auxiliary␣file␣´); print_aux_name; decr(aux_ptr);
    aux_err_return;
    end;
  print(´A␣level-´, aux_ptr : 0, ´␣auxiliary␣file:␣´); print_aux_name; cur_aux_line ← 0;
  end
```

This code is used in section 140.

142. Here we close the current-level `.aux` file and go back up a level, if possible, by decrementing *aux_ptr*.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure pop_the_aux_stack;
  begin a_close(cur_aux_file);
  if (aux_ptr = 0) then goto aux_done
  else decr(aux_ptr);
  end;
```

143. That's it for processing `.aux` commands, except for finishing the procedural gymnastics.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
  ⟨ Scan for and process an .aux command 116 ⟩
```

144. We must complain if anything's amiss.

```
  define aux_end_err(#) ≡
         begin aux_end1_err_print; print(#); aux_end2_err_print;
         end
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure aux_end1_err_print;
  begin print(´I␣found␣no␣´);
  end;
procedure aux_end2_err_print;
  begin print(´---while␣reading␣file␣´); print_aux_name; mark_error;
  end;
```

145. Before proceeding, we see if we have any complaints.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure last_check_for_aux_errors;
  begin num_cites ← cite_ptr; { record the number of distinct cite keys }
  num_bib_files ← bib_ptr; { and the number of .bib files }
  if (¬citation_seen) then aux_end_err(´\citation␣commands´)
  else if ((num_cites = 0) ∧ (¬all_entries)) then aux_end_err(´cite␣keys´);
  if (¬bib_seen) then aux_end_err(´\bibdata␣command´)
  else if (num_bib_files = 0) then aux_end_err(´database␣files´);
  if (¬bst_seen) then aux_end_err(´\bibstyle␣command´)
  else if (bst_str = 0) then aux_end_err(´style␣file´);
  end;
```

146. Reading the style file. This part of the program reads the `.bst` file, which consists of a sequence of commands. Each `.bst` command consists of a name (for which case differences are ignored) followed by zero or more arguments, each enclosed in braces.

define *bst_done* = 32 { go here when finished with the `.bst` file }
define *no_bst_file* = 9932 { go here when skipping the `.bst` file }

⟨ Labels in the outer block 109 ⟩ +≡
, *bst_done*, *no_bst_file*

147. The *bbl_line_num* gets initialized along with the *bst_line_num*, so it's declared here too.

⟨ Globals in the outer block 16 ⟩ +≡
bbl_line_num: *integer*; { line number of the `.bbl` (output) file }
bst_line_num: *integer*; { line number of the `.bst` file }

148. This little procedure exists because it's used by at least two other procedures and thus saves some space.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *bst_ln_num_print*;
begin *print*(´--line␣´, *bst_line_num* : 0, ´␣of␣file␣´); *print_bst_name*;
end;

149. When there's a serious error parsing the `.bst` file, we flush the rest of the current command; a blank line is assumed to mark the end of a command (but for the purposes of error recovery only). Thus, error recovery will be better if style designers leave blank lines between `.bst` commands. This macro must be called from within a procedure that has an *exit* label.

define *bst_err_print_and_look_for_blank_line_return* ≡
begin *bst_err_print_and_look_for_blank_line*; **return**;
end
define *bst_err*(#) ≡
begin { serious error during `.bst` parsing }
print(#); *bst_err_print_and_look_for_blank_line_return*;
end

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *bst_err_print_and_look_for_blank_line*;
begin *print*(´-´); *bst_ln_num_print*; *print_bad_input_line*; { this call does the *mark_error* }
while (*last* ≠ 0) **do** { look for a blank input line }
if (¬*input_ln*(*bst_file*)) **then** { or the end of the file }
goto *bst_done*
else *incr*(*bst_line_num*);
buf_ptr2 ← *last*; { to input the next line }
end;

150. When there's a harmless error parsing the `.bst` file (harmless syntactically, at least) we give just a *warning_message*.

```
define bst_warn(#) ≡
        begin  { non-serious error during .bst parsing }
        print(#); bst_warn_print;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bst_warn_print;
  begin bst_ln_num_print; mark_warning;
  end;
```

151. Here's the outer loop for reading the `.bst` file—it keeps reading and processing `.bst` commands until none left. This is part of the main program; hence, because of the *bst_done* label, there's no conventional **begin - end** pair surrounding the entire module.

⟨ Read and execute the `.bst` file 151 ⟩ ≡

```
  if (bst_str = 0) then  { there's no .bst file to read }
    goto no_bst_file;  { this is a goto so that bst_done is not in a block }
  bst_line_num ← 0;  { initialize things }
  bbl_line_num ← 1;  { best spot to initialize the output line number }
  buf_ptr2 ← last;  { to get the first input line }
  loop
    begin if (¬eat_bst_white_space) then  { the end of the .bst file }
      goto bst_done;
    get_bst_command_and_process;
    end;
bst_done: a_close(bst_file);
no_bst_file: a_close(bbl_file);
```

This code is used in section 10.

152. This `.bst`-specific scanning function skips over *white_space* characters (and comments) until hitting a nonwhite character or the end of the file, respectively returning *true* or *false*. It also updates *bst_line_num*, the line counter.

⟨ Procedures and functions for input scanning 83 ⟩ +≡

```
function eat_bst_white_space: boolean;
  label exit;
  begin loop
    begin if (scan_white_space) then  { hit a nonwhite character on this line }
      if (scan_char ≠ comment) then  { it's not a comment character; return }
        begin eat_bst_white_space ← true; return;
        end;
    if (¬input_ln(bst_file)) then  { end-of-file; return false }
      begin eat_bst_white_space ← false; return;
      end;
    incr(bst_line_num); buf_ptr2 ← 0;
    end;
exit: end;
```

153. It's often illegal to end a `.bst` command in certain places, and this is where we come to check.

```
define eat_bst_white_and_eof_check(#) ≡
        begin if (¬eat_bst_white_space) then
          begin eat_bst_print; bst_err(#);
          end;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure eat_bst_print;
  begin print('Illegal␣end␣of␣style␣file␣in␣command:␣');
  end;
```

154. We must attend to a few details before getting to work on this `.bst` command.

⟨ Scan for and process a `.bst` command 154 ⟩ ≡

```
procedure get_bst_command_and_process;
  label exit;
  begin if (¬scan_alpha) then bst_err('"', xchr[scan_char], '"␣can''t␣start␣a␣style-file␣command');
  lower_case(buffer, buf_ptr1, token_len);  { ignore case differences }
  command_num ← ilk_info[str_lookup(buffer, buf_ptr1, token_len, bst_command_ilk, dont_insert)];
  if (¬hash_found) then
    begin print_token; bst_err('␣is␣an␣illegal␣style-file␣command');
    end;
  ⟨ Process the appropriate .bst command 155 ⟩;
exit: end;
```

This code is used in section 217.

155. Here we determine which `.bst` command we're about to process, and then go to it.

⟨ Process the appropriate `.bst` command 155 ⟩ ≡

```
  case (command_num) of
  n_bst_entry: bst_entry_command;
  n_bst_execute: bst_execute_command;
  n_bst_function: bst_function_command;
  n_bst_integers: bst_integers_command;
  n_bst_iterate: bst_iterate_command;
  n_bst_macro: bst_macro_command;
  n_bst_read: bst_read_command;
  n_bst_reverse: bst_reverse_command;
  n_bst_sort: bst_sort_command;
  n_bst_strings: bst_strings_command;
  othercases confusion('Unknown␣style-file␣command')
  endcases
```

This code is used in section 154.

156. We need data structures for the function definitions, the entry variables, the global variables, and the actual entries corresponding to the cite-key list. First we define the classes of 'function's used. Functions in all classes are of *bst_fn_ilk* except for *int_literal*s, which are of *integer_ilk*; and *str_literal*s, which are of *text_ilk*.

define *built_in* = 0 { the 'primitive' functions }
define *wiz_defined* = 1 { defined in the `.bst` file }
define *int_literal* = 2 { integer 'constants' }
define *str_literal* = 3 { string 'constants' }
define *field* = 4 { things like 'author' and 'title' }
define *int_entry_var* = 5 { integer entry variable }
define *str_entry_var* = 6 { string entry variable }
define *int_global_var* = 7 { integer global variable }
define *str_global_var* = 8 { string global variable }
define *last_fn_class* = 8 { the same number as on the line above }

157. Here's another bug report.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *unknwn_function_class_confusion*;
begin *confusion*(´Unknown␣function␣class´);
end;

158. Occasionally we'll want to *print* the name of one of these function classes.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *print_fn_class*(*fn_loc* : *hash_loc*);
begin case (*fn_type*[*fn_loc*]) **of**
built_in: *print*(´built-in´);
wiz_defined: *print*(´wizard-defined´);
int_literal: *print*(´integer-literal´);
str_literal: *print*(´string-literal´);
field: *print*(´field´);
int_entry_var: *print*(´integer-entry-variable´);
str_entry_var: *print*(´string-entry-variable´);
int_global_var: *print*(´integer-global-variable´);
str_global_var: *print*(´string-global-variable´);
othercases *unknwn_function_class_confusion*
endcases;
end;

159. This version is for printing when in **trace** mode.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
  trace procedure trace_pr_fn_class(fn_loc : hash_loc);
  begin case (fn_type[fn_loc]) of
  built_in: trace_pr('built-in');
  wiz_defined: trace_pr('wizard-defined');
  int_literal: trace_pr('integer-literal');
  str_literal: trace_pr('string-literal');
  field: trace_pr('field');
  int_entry_var: trace_pr('integer-entry-variable');
  str_entry_var: trace_pr('string-entry-variable');
  int_global_var: trace_pr('integer-global-variable');
  str_global_var: trace_pr('string-global-variable');
  othercases unknwn_function_class_confusion
  endcases;
  end;
  ecart
```

160. Besides the function classes, we have types based on BibTeX's capacity limitations and one based on what can go into the array *wiz_functions* explained below.

```
  define quote_next_fn = hash_base − 1  { special marker used in defining functions }
  define end_of_def = hash_max + 1  { another such special marker }
⟨ Types in the outer block 22 ⟩ +≡
  fn_class = 0 .. last_fn_class;  { the .bst function classes }
  wiz_fn_loc = 0 .. wiz_fn_space;  { wiz_defined-function storage locations }
  int_ent_loc = 0 .. max_ent_ints;  { int_entry_var storage locations }
  str_ent_loc = 0 .. max_ent_strs;  { str_entry_var storage locations }
  str_glob_loc = 0 .. max_glb_str_minus_1;  { str_global_var storage locations }
  field_loc = 0 .. max_fields;  { individual field storage locations }
  hash_ptr2 = quote_next_fn .. end_of_def;  { a special marker or a hash_loc }
```

161. We store information about the `.bst` functions in arrays the same size as the hash-table arrays and in locations corresponding to their hash-table locations. The two arrays *fn_info* (an alias of *ilk_info* described earlier) and *fn_type* accomplish this: *fn_type* specifies one of the above classes, and *fn_info* gives information dependent on the class.

Six other arrays give the contents of functions: The array *wiz_functions* holds definitions for *wiz_defined* functions—each such function consists of a sequence of pointers to hash-table locations of other functions (with the two special-marker exceptions above); the array *entry_ints* contains the current values of *int_entry_var*s; the array *entry_strs* contains the current values of *str_entry_var*s; an element of the array *global_strs* contains the current value of a *str_global_var* if the corresponding *glb_str_ptr* entry is empty, otherwise the nonempty entry is a pointer to the string; and the array *field_info*, for each field of each entry, contains either a pointer to the string or the special value *missing*.

The array *global_strs* isn't packed (that is, it isn't **array** ... **of packed array** ...) to increase speed on some systems; however, on systems that are byte-addressable and that have a good compiler, packing *global_strs* would save lots of space without much loss of speed.

define *fn_info* ≡ *ilk_info* { an alias used with functions }

define *missing* = *empty* { a special pointer for missing fields }

⟨ Globals in the outer block 16 ⟩ +≡
fn_loc: *hash_loc*; { the hash-table location of a function }
wiz_loc: *hash_loc*; { the hash-table location of a wizard function }
literal_loc: *hash_loc*; { the hash-table location of a literal function }
macro_name_loc: *hash_loc*; { the hash-table location of a macro name }
macro_def_loc: *hash_loc*; { the hash-table location of a macro definition }
fn_type: **packed array** [*hash_loc*] **of** *fn_class*;
wiz_def_ptr: *wiz_fn_loc*; { storage location for the next wizard function }
wiz_fn_ptr: *wiz_fn_loc*; { general *wiz_functions* location }
wiz_functions: **packed array** [*wiz_fn_loc*] **of** *hash_ptr2*;
int_ent_ptr: *int_ent_loc*; { general *int_entry_var* location }
entry_ints: **array** [*int_ent_loc*] **of** *integer*;
num_ent_ints: *int_ent_loc*; { the number of distinct *int_entry_var* names }
str_ent_ptr: *str_ent_loc*; { general *str_entry_var* location }
entry_strs: **array** [*str_ent_loc*] **of packed array** [0 .. *ent_str_size*] **of** *ASCII_code*;
num_ent_strs: *str_ent_loc*; { the number of distinct *str_entry_var* names }
str_glb_ptr: 0 .. *max_glob_strs*; { general *str_global_var* location }
glb_str_ptr: **array** [*str_glob_loc*] **of** *str_number*;
global_strs: **array** [*str_glob_loc*] **of array** [0 .. *glob_str_size*] **of** *ASCII_code*;
glb_str_end: **array** [*str_glob_loc*] **of** 0 .. *glob_str_size*; { end markers }
num_glb_strs: 0 .. *max_glob_strs*; { number of distinct *str_global_var* names }
field_ptr: *field_loc*; { general *field_info* location }
field_parent_ptr, *field_end_ptr*: *field_loc*; { two more for doing cross-refs }
cite_parent_ptr, *cite_xptr*: *cite_number*; { two others for doing cross-refs }
field_info: **packed array** [*field_loc*] **of** *str_number*;
num_fields: *field_loc*; { the number of distinct field names }
num_pre_defined_fields: *field_loc*; { so far, just one: `crossref` }
crossref_num: *field_loc*; { the number given to `crossref` }
no_fields: *boolean*; { used for *tr_print*ing entry information }

162. Now we initialize storage for the *wiz_defined* functions and we initialize variables so that the first *str_entry_var*, *int_entry_var*, *str_global_var*, and *field* name will be assigned the number 0. Note: The variables *num_ent_strs* and *num_fields* will also be set when pre-defining strings.

```
⟨ Set initial values of key variables 20 ⟩ +≡
  wiz_def_ptr ← 0; num_ent_ints ← 0; num_ent_strs ← 0; num_fields ← 0; str_glb_ptr ← 0;
  while (str_glb_ptr < max_glob_strs) do { make str_global_vars empty }
    begin glb_str_ptr[str_glb_ptr] ← 0; glb_str_end[str_glb_ptr] ← 0; incr(str_glb_ptr);
    end;
  num_glb_strs ← 0;
```

163. Style-file commands. There are ten `.bst` commands: Five (`entry`, `function`, `integers`, `macro`, and `strings`) declare and define functions, one (`read`) reads in the `.bib`-file entries, and four (`execute`, `iterate`, `reverse`, and `sort`) manipulate the entries and produce output.

The boolean variables *entry_seen* and *read_seen* indicate whether we've yet encountered an `entry` and a `read` command. There must be exactly one of each of these, and the `entry` command, as well as any `macro` command, must precede the `read` command. Furthermore, the `read` command must precede the four that manipulate the entries and produce output.

```
⟨ Globals in the outer block 16 ⟩ +≡
entry_seen: boolean;  { true if we've already seen an entry command }
read_seen: boolean;  { true if we've already seen a read command }
read_performed: boolean;  { true if we started reading the database file(s) }
reading_completed: boolean;  { true if we made it all the way through }
read_completed: boolean;  { true if the database info didn't bomb BibTeX }
```

164. And we initialize them.

```
⟨ Set initial values of key variables 20 ⟩ +≡
  entry_seen ← false; read_seen ← false; read_performed ← false; reading_completed ← false;
  read_completed ← false;
```

165. Here's another bug.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure id_scanning_confusion;
  begin confusion('Identifier␣scanning␣error');
  end;
```

166. This macro is used to scan all `.bst` identifiers. The argument supplies the `.bst` command name. The associated procedure simply prints an error message.

```
  define bst_identifier_scan(#) ≡
          begin scan_identifier(right_brace, comment, comment);
          if ((scan_result = white_adjacent) ∨ (scan_result = specified_char_adjacent)) then do_nothing
          else begin bst_id_print; bst_err(#);
            end;
          end
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure bst_id_print;
  begin if (scan_result = id_null) then print('"', xchr[scan_char], '"␣begins␣identifier,␣command:␣')
  else if (scan_result = other_char_adjacent) then
      print('"', xchr[scan_char], '"␣immediately␣follows␣identifier,␣command:␣')
    else id_scanning_confusion;
  end;
```

167. This macro just makes sure we're at a *left_brace*.

```
define bst_get_and_check_left_brace(#) ≡
        begin if (scan_char ≠ left_brace) then
          begin bst_left_brace_print; bst_err(#);
          end;
        incr(buf_ptr2);  { skip over the left_brace }
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bst_left_brace_print;
  begin print('"', xchr[left_brace], '" is missing in command: ');
  end;
```

168. And this one, a *right_brace*.

```
define bst_get_and_check_right_brace(#) ≡
        begin if (scan_char ≠ right_brace) then
          begin bst_right_brace_print; bst_err(#);
          end;
        incr(buf_ptr2);  { skip over the right_brace }
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bst_right_brace_print;
  begin print('"', xchr[right_brace], '" is missing in command: ');
  end;
```

169. This macro complains if we've already encountered a function to be inserted into the hash table.

```
define check_for_already_seen_function(#) ≡
        begin if (hash_found) then  { already encountered this as a .bst function }
          begin already_seen_function_print(#); return;
          end;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure already_seen_function_print(seen_fn_loc : hash_loc);
  label exit;  { so the call to bst_err works }
  begin print_pool_str(hash_text[seen_fn_loc]); print(' is already a type "');
  print_fn_class(seen_fn_loc); print_ln('" function name'); bst_err_print_and_look_for_blank_line_return;
exit: end;
```

170. An `entry` command has three arguments, each a (possibly empty) list of function names between braces (the names are separated by one or more *white_space* characters). All function names in this and other commands must be legal `.bst` identifiers. Upper/lower cases are considered to be the same for function names in these lists—all upper-case letters are converted to lower case. These arguments give lists of *field*s, *int_entry_var*s, and *str_entry_var*s.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_entry_command;
  label exit;
  begin if (entry_seen) then bst_err('Illegal, another entry command');
  entry_seen ← true;  { now we've seen an entry command }
  eat_bst_white_and_eof_check('entry'); ⟨ Scan the list of fields 171 ⟩;
  eat_bst_white_and_eof_check('entry');
  if (num_fields = num_pre_defined_fields) then bst_warn('Warning--I didn't find any fields');
  ⟨ Scan the list of int_entry_vars 173 ⟩;
  eat_bst_white_and_eof_check('entry'); ⟨ Scan the list of str_entry_vars 175 ⟩;
exit: end;
```

171. This module reads a *left_brace*, the list of *field*s, and a *right_brace*. The *field*s are those like 'author' and 'title.'

```
⟨ Scan the list of fields 171 ⟩ ≡
  begin bst_get_and_check_left_brace('entry'); eat_bst_white_and_eof_check('entry');
  while (scan_char ≠ right_brace) do
    begin bst_identifier_scan('entry'); ⟨ Insert a field into the hash table 172 ⟩;
    eat_bst_white_and_eof_check('entry');
    end;
  incr(buf_ptr2);  { skip over the right_brace }
  end
```

This code is used in section 170.

172. Here we insert the just found field name into the hash table, record it as a *field*, and assign it a number to be used in indexing into the *field_info* array.

```
⟨ Insert a field into the hash table 172 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln(' is a field');
  ecart
  lower_case(buffer, buf_ptr1, token_len);  { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, do_insert);
  check_for_already_seen_function(fn_loc); fn_type[fn_loc] ← field;
  fn_info[fn_loc] ← num_fields;  { give this field a number (take away its name) }
  incr(num_fields);
  end
```

This code is used in section 171.

173. This module reads a *left_brace*, the list of *int_entry_var*s, and a *right_brace*.

```
⟨ Scan the list of int_entry_vars 173 ⟩ ≡
  begin bst_get_and_check_left_brace(´entry´); eat_bst_white_and_eof_check(´entry´);
  while (scan_char ≠ right_brace) do
    begin bst_identifier_scan(´entry´); ⟨ Insert an int_entry_var into the hash table 174 ⟩;
    eat_bst_white_and_eof_check(´entry´);
    end;
  incr(buf_ptr2); { skip over the right_brace }
  end
```

This code is used in section 170.

174. Here we insert the just found *int_entry_var* name into the hash table and record it as an *int_entry_var*. An *int_entry_var* is one that the style designer wants a separate copy of for each entry.

```
⟨ Insert an int_entry_var into the hash table 174 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln(´␣is␣an␣integer␣entry-variable´);
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, do_insert);
  check_for_already_seen_function(fn_loc); fn_type[fn_loc] ← int_entry_var;
  fn_info[fn_loc] ← num_ent_ints; { give this int_entry_var a number }
  incr(num_ent_ints);
  end
```

This code is used in section 173.

175. This module reads a *left_brace*, the list of *str_entry_var*s, and a *right_brace*. A *str_entry_var* is one that the style designer wants a separate copy of for each entry.

```
⟨ Scan the list of str_entry_vars 175 ⟩ ≡
  begin bst_get_and_check_left_brace(´entry´); eat_bst_white_and_eof_check(´entry´);
  while (scan_char ≠ right_brace) do
    begin bst_identifier_scan(´entry´); ⟨ Insert a str_entry_var into the hash table 176 ⟩;
    eat_bst_white_and_eof_check(´entry´);
    end;
  incr(buf_ptr2); { skip over the right_brace }
  end
```

This code is used in section 170.

176. Here we insert the just found *str_entry_var* name into the hash table, record it as a *str_entry_var*, and set its pointer into *entry_strs*.

```
⟨ Insert a str_entry_var into the hash table 176 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln(´␣is␣a␣string␣entry-variable´);
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, do_insert);
  check_for_already_seen_function(fn_loc); fn_type[fn_loc] ← str_entry_var;
  fn_info[fn_loc] ← num_ent_strs; { give this str_entry_var a number }
  incr(num_ent_strs);
  end
```

This code is used in section 175.

177. A legal argument for an `execute`, `iterate`, or `reverse` command must exist and be *built_in* or *wiz_defined*. Here's where we check, returning *true* if the argument is illegal.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
function bad_argument_token: boolean;
  label exit;
  begin bad_argument_token ← true;  { now it's easy to exit if necessary }
  lower_case(buffer, buf_ptr1, token_len);  { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, dont_insert);
  if (¬hash_found) then  { unknown .bst function }
    begin print_token; bst_err(´␣is␣an␣unknown␣function´);
    end
  else if ((fn_type[fn_loc] ≠ built_in) ∧ (fn_type[fn_loc] ≠ wiz_defined)) then
      begin print_token; print(´␣has␣bad␣function␣type␣´); print_fn_class(fn_loc);
      bst_err_print_and_look_for_blank_line_return;
      end;
  bad_argument_token ← false;
exit: end;
```

178. An `execute` command has one argument, a single *built_in* or *wiz_defined* function name between braces. Upper/lower cases are considered to be the same—all upper-case letters are converted to lower case. Also, we must make sure we've already seen a `read` command.

This module reads a *left_brace*, a single function to be executed, and a *right_brace*.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_execute_command;
  label exit;
  begin if (¬read_seen) then bst_err(´Illegal,␣execute␣command␣before␣read␣command´);
  eat_bst_white_and_eof_check(´execute´); bst_get_and_check_left_brace(´execute´);
  eat_bst_white_and_eof_check(´execute´); bst_identifier_scan(´execute´);
  ⟨ Check the execute-command argument token 179 ⟩;
  eat_bst_white_and_eof_check(´execute´); bst_get_and_check_right_brace(´execute´);
  ⟨ Perform an execute command 296 ⟩;
exit: end;
```

179. Before executing the function, we must make sure it's a legal one. It must exist and be *built_in* or *wiz_defined*.

```
⟨ Check the execute-command argument token 179 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln(´␣is␣a␣to␣be␣executed␣function´);
  ecart
  if (bad_argument_token) then return;
  end
```

This code is used in section 178.

180. A `function` command has two arguments; the first is a *wiz_defined* function name between braces. Upper/lower cases are considered to be the same—all upper-case letters are converted to lower case. The second argument defines this function. It consists of a sequence of functions, between braces, separated by *white_space* characters. Upper/lower cases are considered to be the same for function names but not for *str_literal*s.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_function_command;
  label exit;
  begin eat_bst_white_and_eof_check(´function´); ⟨ Scan the wiz_defined function name 181 ⟩;
  eat_bst_white_and_eof_check(´function´); bst_get_and_check_left_brace(´function´);
  scan_fn_def(wiz_loc); { this scans the function definition }
exit: end;
```

181. This module reads a *left_brace*, a *wiz_defined* function name, and a *right_brace*.

```
⟨ Scan the wiz_defined function name 181 ⟩ ≡
  begin bst_get_and_check_left_brace(´function´); eat_bst_white_and_eof_check(´function´);
  bst_identifier_scan(´function´); ⟨ Check the wiz_defined function name 182 ⟩;
  eat_bst_white_and_eof_check(´function´); bst_get_and_check_right_brace(´function´);
  end
```

This code is used in section 180.

182. The function name must exist and be a new one; we mark it as *wiz_defined*. Also, see if it's the default entry-type function.

```
⟨ Check the wiz_defined function name 182 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln(´␣is␣a␣wizard-defined␣function´);
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  wiz_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, do_insert);
  check_for_already_seen_function(wiz_loc); fn_type[wiz_loc] ← wiz_defined;
  if (hash_text[wiz_loc] = s_default) then { we've found the default entry-type }
    b_default ← wiz_loc; { see the built_in functions for b_default }
  end
```

This code is used in section 181.

183. We're about to start scanning tokens in a function definition. When a function token is illegal, we skip until it ends; a *white_space* character, an end-of-line, a *right_brace*, or a *comment* marks the end of the current token.

```
  define next_token = 25 { a bad function token; go read the next one }
  define skip_token(#) ≡
          begin { not-so-serious error during .bst parsing }
          print(#); skip_token_print; { also, skip to the current token's end }
          goto next_token;
          end
⟨ Procedures and functions for input scanning 83 ⟩ +≡
procedure skip_token_print;
  begin print(´-´); bst_ln_num_print; mark_error;
  if (scan2_white(right_brace, comment)) then { ok if token ends line }
    do_nothing;
  end;
```

184. This macro is similar to the last one but is specifically for recursion in a *wiz_defined* function, which is illegal; it helps save space.

```
define skip_recursive_token ≡
        begin print_recursion_illegal; goto next_token;
        end
```

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
procedure print_recursion_illegal;
  begin trace trace_pr_newline;
  ecart
  print_ln('Curse␣you,␣wizard,␣before␣you␣recurse␣me:'); print('function␣'); print_token;
  print_ln('␣is␣illegal␣in␣its␣own␣definition'); @{print_recursion_illegal; @}
  skip_token_print;  { also, skip to the current token's end }
  end;
```

185. Here's another macro for saving some space when there's a problem with a token.

```
define skip_token_unknown_function ≡
        begin skp_token_unknown_function_print; goto next_token;
        end
```

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
procedure skp_token_unknown_function_print;
  begin print_token; print('␣is␣an␣unknown␣function'); skip_token_print;
      { also, skip to the current token's end }
  end;
```

186. And another.

```
define skip_token_illegal_stuff_after_literal ≡
        begin skip_illegal_stuff_after_token_print; goto next_token;
        end
```

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
procedure skip_illegal_stuff_after_token_print;
  begin print('"', xchr[scan_char], '"␣can''t␣follow␣a␣literal'); skip_token_print;
      { also, skip to the current token's end }
  end;
```

187. This recursive function reads and stores the list of functions (separated by *white_space* characters or ends-of-line) that define this new function, and reads a *right_brace*.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
procedure scan_fn_def(fn_hash_loc : hash_loc);
  label next_token, exit;
  type fn_def_loc = 0 .. single_fn_space;  { for a single wiz_defined-function }
  var singl_function: packed array [fn_def_loc] of hash_ptr2;
    single_ptr: fn_def_loc;  { next storage location for this definition }
    copy_ptr: fn_def_loc;  { dummy variable }
    end_of_num: buf_pointer;  { the end of an implicit function's name }
    impl_fn_loc: hash_loc;  { an implicit function's hash-table location }
  begin eat_bst_white_and_eof_check('function'); single_ptr ← 0;
  while (scan_char ≠ right_brace) do
    begin ⟨ Get the next function of the definition 189 ⟩;
  next_token: eat_bst_white_and_eof_check('function');
    end;
  ⟨ Complete this function's definition 200 ⟩;
  incr(buf_ptr2);  { skip over the right_brace }
exit: end;
```

188. This macro inserts a hash-table location (or one of the two special markers *quote_next_fn* and *end_of_def*) into the *singl_function* array, which will later be copied into the *wiz_functions* array.

```
define insert_fn_loc(#) ≡
        begin singl_function[single_ptr] ← #;
        if (single_ptr = single_fn_space) then singl_fn_overflow;
        incr(single_ptr);
        end

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure singl_fn_overflow;
  begin overflow('single␣function␣space␣', single_fn_space);
  end;
```

189. There are five possibilities for the first character of the token representing the next function of the definition: If it's a *number_sign*, the token is an *int_literal*; if it's a *double_quote*, the token is a *str_literal*; if it's a *single_quote*, the token is a quoted function; if it's a *left_brace*, the token isn't really a token, but rather the start of another function definition (which will result in a recursive call to *scan_fn_def*); if it's anything else, the token is the name of an already-defined function. Note: To prevent the wizard from using recursion, we have to check that neither a quoted function nor an already-defined-function is actually the currently-being-defined function (which is stored at *wiz_loc*).

```
⟨ Get the next function of the definition 189 ⟩ ≡
  case (scan_char) of
  number_sign: ⟨ Scan an int_literal 190 ⟩;
  double_quote: ⟨ Scan a str_literal 191 ⟩;
  single_quote: ⟨ Scan a quoted function 192 ⟩;
  left_brace: ⟨ Start a new function definition 194 ⟩;
  othercases ⟨ Scan an already-defined function 199 ⟩
  endcases
```

This code is used in section 187.

190. An *int_literal* is preceded by a *number_sign*, consists of an integer (i.e., an optional *minus_sign* followed by one or more *numeric* characters), and is followed either by a *white_space* character, an end-of-line, or a *right_brace*. The array *fn_info* contains the value of the integer for *int_literal*s.

```
⟨ Scan an int_literal 190 ⟩ ≡
  begin incr(buf_ptr2); { skip over the number_sign }
  if (¬scan_integer) then skip_token('Illegal␣integer␣in␣integer␣literal');
  trace trace_pr('#'); trace_pr_token;
  trace_pr_ln('␣is␣an␣integer␣literal␣with␣value␣', token_value : 0);
  ecart
  literal_loc ← str_lookup(buffer, buf_ptr1, token_len, integer_ilk, do_insert);
  if (¬hash_found) then
    begin fn_type[literal_loc] ← int_literal; { set the fn_class }
    fn_info[literal_loc] ← token_value; { the value of this integer }
    end;
  if ((lex_class[scan_char] ≠ white_space) ∧ (buf_ptr2 < last) ∧ (scan_char ≠ right_brace) ∧
        (scan_char ≠ comment)) then skip_token_illegal_stuff_after_literal;
  insert_fn_loc(literal_loc); { add this function to wiz_functions }
  end
```

This code is used in section 189.

191. A *str_literal* is preceded by a *double_quote* and consists of all characters on this line up to the next *double_quote*. Also, there must be either a *white_space* character, an end-of-line, a *right_brace*, or a *comment* following (since functions in the definition must be separated by *white_space*). The array *fn_info* contains nothing for *str_literal*s.

```
⟨ Scan a str_literal 191 ⟩ ≡
  begin incr(buf_ptr2); { skip over the double_quote }
  if (¬scan1(double_quote)) then skip_token('No␣`', xchr[double_quote], '␣to␣end␣string␣literal');
  trace trace_pr('"'); trace_pr_token; trace_pr('"'); trace_pr_ln('␣is␣a␣string␣literal');
  ecart
  literal_loc ← str_lookup(buffer, buf_ptr1, token_len, text_ilk, do_insert);
  fn_type[literal_loc] ← str_literal; { set the fn_class }
  incr(buf_ptr2); { skip over the double_quote }
  if ((lex_class[scan_char] ≠ white_space) ∧ (buf_ptr2 < last) ∧ (scan_char ≠ right_brace) ∧
        (scan_char ≠ comment)) then skip_token_illegal_stuff_after_literal;
  insert_fn_loc(literal_loc); { add this function to wiz_functions }
  end
```

This code is used in section 189.

192. A quoted function is preceded by a *single_quote* and consists of all characters up to the next *white_space* character, end-of-line, *right_brace*, or *comment*.

```
⟨ Scan a quoted function 192 ⟩ ≡
  begin incr(buf_ptr2); { skip over the single_quote }
  if (scan2_white(right_brace, comment)) then { ok if token ends line }
    do_nothing;
  trace trace_pr(´´´´); trace_pr_token; trace_pr(´␣is␣a␣quoted␣function␣´);
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, dont_insert);
  if (¬hash_found) then { unknown .bst function }
    skip_token_unknown_function
  else ⟨ Check and insert the quoted function 193 ⟩;
  end
```

This code is used in section 189.

193. Here we check that this quoted function is a legal one—the function name must already exist, but it mustn't be the currently-being-defined function (which is stored at *wiz_loc*).

```
⟨ Check and insert the quoted function 193 ⟩ ≡
  begin if (fn_loc = wiz_loc) then skip_recursive_token
  else begin trace trace_pr(´of␣type␣´); trace_pr_fn_class(fn_loc); trace_pr_newline;
    ecart
    insert_fn_loc(quote_next_fn); { add special marker together with }
    insert_fn_loc(fn_loc); { this function to wiz_functions }
    end
  end
```

This code is used in section 192.

194. This module marks the implicit function as being quoted, generates a name, and stores it in the hash table. This name is strictly internal to this program, starts with a *single_quote* (since that will make this function name unique), and ends with the variable *impl_fn_num* converted to ASCII. The alias kludge helps make the stack space not overflow on some machines.

```
  define ex_buf2 ≡ ex_buf { an alias, used only in this module }

⟨ Start a new function definition 194 ⟩ ≡
  begin ex_buf2[0] ← single_quote; int_to_ASCII(impl_fn_num, ex_buf2, 1, end_of_num);
  impl_fn_loc ← str_lookup(ex_buf2, 0, end_of_num, bst_fn_ilk, do_insert);
  if (hash_found) then confusion(´Already␣encountered␣implicit␣function´);
  trace trace_pr_pool_str(hash_text[impl_fn_loc]); trace_pr_ln(´␣is␣an␣implicit␣function´);
  ecart
  incr(impl_fn_num); fn_type[impl_fn_loc] ← wiz_defined;
  insert_fn_loc(quote_next_fn); { all implicit functions are quoted }
  insert_fn_loc(impl_fn_loc); { add it to wiz_functions }
  incr(buf_ptr2); { skip over the left_brace }
  scan_fn_def(impl_fn_loc); { this is the recursive call }
  end
```

This code is used in section 189.

195. The variable *impl_fn_num* counts the number of implicit functions seen in the `.bst` file.

```
⟨ Globals in the outer block 16 ⟩ +≡
impl_fn_num: integer; { the number of implicit functions seen so far }
```

196. Now we initialize it.

```
⟨ Set initial values of key variables 20 ⟩ +≡
  impl_fn_num ← 0;
```

197. This module appends a character to *int_buf* after checking to make sure it will fit; for use in *int_to_ASCII*.

```
define append_int_char(#) ≡
       begin if (int_ptr = buf_size) then buffer_overflow;
       int_buf[int_ptr] ← #; incr(int_ptr);
       end
```

198. This procedure takes the integer *int*, copies the appropriate *ASCII_code* string into *int_buf* starting at *int_begin*, and sets the **var** parameter *int_end* to the first unused *int_buf* location. The ASCII string will consist of decimal digits, the first of which will be not be a 0 if the integer is nonzero, with a prepended minus sign if the integer is negative.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure int_to_ASCII(int : integer; var int_buf : buf_type; int_begin : buf_pointer;
        var int_end : buf_pointer);
  var int_ptr, int_xptr: buf_pointer;   { pointers into int_buf }
    int_tmp_val: ASCII_code;   { the temporary element in an exchange }
  begin int_ptr ← int_begin;
  if (int < 0) then   { add the minus_sign and use the absolute value }
    begin append_int_char(minus_sign); int ← −int;
    end;
  int_xptr ← int_ptr;
  repeat   { copy digits into int_buf }
    append_int_char("0" + (int mod 10)); int ← int div 10;
  until (int = 0);
  int_end ← int_ptr;   { set the string length }
  decr(int_ptr);
  while (int_xptr < int_ptr) do   { and reorder (flip) the digits }
    begin int_tmp_val ← int_buf[int_xptr]; int_buf[int_xptr] ← int_buf[int_ptr];
    int_buf[int_ptr] ← int_tmp_val; decr(int_ptr); incr(int_xptr);
    end
  end;
```

199. An already-defined function consists of all characters up to the next *white_space* character, end-of-line, *right_brace*, or *comment*. This function name must already exist, but it mustn't be the currently-being-defined function (which is stored at *wiz_loc*).

```
⟨ Scan an already-defined function 199 ⟩ ≡
  begin if (scan2_white(right_brace, comment)) then { ok if token ends line }
    do_nothing;
  trace trace_pr_token; trace_pr('␣is␣a␣function␣');
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, dont_insert);
  if (¬hash_found) then { unknown .bst function }
    skip_token_unknown_function
  else if (fn_loc = wiz_loc) then skip_recursive_token
    else begin trace trace_pr('of␣type␣'); trace_pr_fn_class(fn_loc); trace_pr_newline;
      ecart
      insert_fn_loc(fn_loc); { add this function to wiz_functions }
      end;
  end
```

This code is used in section 189.

200. Now we add the *end_of_def* special marker, make sure this function will fit into *wiz_functions*, and put it there.

```
⟨ Complete this function's definition 200 ⟩ ≡
  begin insert_fn_loc(end_of_def); { add special marker ending the definition }
  if (single_ptr + wiz_def_ptr > wiz_fn_space) then
    begin print(single_ptr + wiz_def_ptr : 0, ':␣');
    overflow('wizard-defined␣function␣space␣', wiz_fn_space);
    end;
  fn_info[fn_hash_loc] ← wiz_def_ptr; { pointer into wiz_functions }
  copy_ptr ← 0;
  while (copy_ptr < single_ptr) do { make this function official }
    begin wiz_functions[wiz_def_ptr] ← singl_function[copy_ptr]; incr(copy_ptr); incr(wiz_def_ptr);
    end;
  end
```

This code is used in section 187.

201. An `integers` command has one argument, a list of function names between braces (the names are separated by one or more *white_space* characters). Upper/lower cases are considered to be the same for function names in these lists—all upper-case letters are converted to lower case. Each name in this list specifies an *int_global_var*. There may be several `integers` commands in the `.bst` file.

This module reads a *left_brace*, a list of *int_global_var*s, and a *right_brace*.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_integers_command;
  label exit;
  begin eat_bst_white_and_eof_check(´integers´); bst_get_and_check_left_brace(´integers´);
  eat_bst_white_and_eof_check(´integers´);
  while (scan_char ≠ right_brace) do
    begin bst_identifier_scan(´integers´); ⟨ Insert an int_global_var into the hash table 202 ⟩;
    eat_bst_white_and_eof_check(´integers´);
    end;
  incr(buf_ptr2); { skip over the right_brace }
exit: end;
```

202. Here we insert the just found *int_global_var* name into the hash table and record it as an *int_global_var*. Also, we initialize it by setting *fn_info*[*fn_loc*] to 0.

```
⟨ Insert an int_global_var into the hash table 202 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln(´␣is␣an␣integer␣global-variable´);
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, do_insert);
  check_for_already_seen_function(fn_loc); fn_type[fn_loc] ← int_global_var;
  fn_info[fn_loc] ← 0; { initialize }
  end
```

This code is used in section 201.

203. An `iterate` command has one argument, a single *built_in* or *wiz_defined* function name between braces. Upper/lower cases are considered to be the same—all upper-case letters are converted to lower case. Also, we must make sure we've already seen a `read` command.

This module reads a *left_brace*, a single function to be iterated, and a *right_brace*.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_iterate_command;
  label exit;
  begin if (¬read_seen) then bst_err(´Illegal,␣iterate␣command␣before␣read␣command´);
  eat_bst_white_and_eof_check(´iterate´); bst_get_and_check_left_brace(´iterate´);
  eat_bst_white_and_eof_check(´iterate´); bst_identifier_scan(´iterate´);
  ⟨ Check the iterate-command argument token 204 ⟩;
  eat_bst_white_and_eof_check(´iterate´); bst_get_and_check_right_brace(´iterate´);
  ⟨ Perform an iterate command 297 ⟩;
exit: end;
```

204. Before iterating the function, we must make sure it's a legal one. It must exist and be *built_in* or *wiz_defined*.

```
⟨ Check the iterate-command argument token 204 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln('␣is␣a␣to␣be␣iterated␣function');
  ecart
  if (bad_argument_token) then return;
  end
```

This code is used in section 203.

205. A `macro` command, like a `function` command, has two arguments; the first is a macro name between braces. The name must be a legal `.bst` identifier. Upper/lower cases are considered to be the same—all upper-case letters are converted to lower case. The second argument defines this macro. It consists of a *double_quote*-delimited string (which must be on a single line) between braces, with optional *white_space* characters between the braces and the *double_quote*s. This *double_quote*-delimited string is parsed exactly as a *str_literal* is for the `function` command.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_macro_command;
  label exit;
  begin if (read_seen) then bst_err('Illegal,␣macro␣command␣after␣read␣command');
  eat_bst_white_and_eof_check('macro'); ⟨ Scan the macro name 206 ⟩;
  eat_bst_white_and_eof_check('macro'); ⟨ Scan the macro's definition 208 ⟩;
exit: end;
```

206. This module reads a *left_brace*, a macro name, and a *right_brace*.

```
⟨ Scan the macro name 206 ⟩ ≡
  begin bst_get_and_check_left_brace('macro'); eat_bst_white_and_eof_check('macro');
  bst_identifier_scan('macro'); ⟨ Check the macro name 207 ⟩;
  eat_bst_white_and_eof_check('macro'); bst_get_and_check_right_brace('macro');
  end
```

This code is used in section 205.

207. The macro name must be a new one; we mark it as *macro_ilk*.

```
⟨ Check the macro name 207 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln('␣is␣a␣macro');
  ecart
  lower_case(buffer, buf_ptr1, token_len);   { ignore case differences }
  macro_name_loc ← str_lookup(buffer, buf_ptr1, token_len, macro_ilk, do_insert);
  if (hash_found) then
    begin print_token; bst_err('␣is␣already␣defined␣as␣a␣macro');
    end;
  ilk_info[macro_name_loc] ← hash_text[macro_name_loc];   { default in case of error }
  end
```

This code is used in section 206.

208. This module reads a *left_brace*, the *double_quote*-delimited string that defines this macro, and a *right_brace*.

```
⟨ Scan the macro's definition 208 ⟩ ≡
  begin bst_get_and_check_left_brace(´macro´); eat_bst_white_and_eof_check(´macro´);
  if (scan_char ≠ double_quote) then
    bst_err(´A␣macro␣definition␣must␣be␣´, xchr[double_quote], ´-delimited´);
  ⟨ Scan the macro definition-string 209 ⟩;
  eat_bst_white_and_eof_check(´macro´); bst_get_and_check_right_brace(´macro´);
  end
```

This code is used in section 205.

209. A macro definition-string is preceded by a *double_quote* and consists of all characters on this line up to the next *double_quote*. The array *ilk_info* contains a pointer to this string for the macro name.

```
⟨ Scan the macro definition-string 209 ⟩ ≡
  begin incr(buf_ptr2);  { skip over the double_quote }
  if (¬scan1(double_quote)) then
    bst_err(´There´´s␣no␣`´, xchr[double_quote], ´´´␣to␣end␣macro␣definition´);
  trace trace_pr(´"´); trace_pr_token; trace_pr(´"´); trace_pr_ln(´␣is␣a␣macro␣string´);
  ecart
  macro_def_loc ← str_lookup(buffer, buf_ptr1, token_len, text_ilk, do_insert);
  fn_type[macro_def_loc] ← str_literal;  { set the fn_class }
  ilk_info[macro_name_loc] ← hash_text[macro_def_loc]; incr(buf_ptr2);  { skip over the double_quote }
  end
```

This code is used in section 208.

210. We need to include stuff for `.bib` reading here because that's done by the `read` command.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
  ⟨ Scan for and process a .bib command or database entry 236 ⟩
```

211. The `read` command has no arguments so there's no more parsing to do. We must make sure we haven't seen a `read` command before and we've already seen an `entry` command.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_read_command;
  label exit;
  begin if (read_seen) then bst_err(´Illegal,␣another␣read␣command´);
  read_seen ← true;  { now we've seen a read command }
  if (¬entry_seen) then bst_err(´Illegal,␣read␣command␣before␣entry␣command´);
  sv_ptr1 ← buf_ptr2;  { save the contents of the .bst input line }
  sv_ptr2 ← last; tmp_ptr ← sv_ptr1;
  while (tmp_ptr < sv_ptr2) do
    begin sv_buffer[tmp_ptr] ← buffer[tmp_ptr]; incr(tmp_ptr);
    end;
  ⟨ Read the .bib file(s) 223 ⟩;
  buf_ptr2 ← sv_ptr1;  { and restore }
  last ← sv_ptr2; tmp_ptr ← buf_ptr2;
  while (tmp_ptr < last) do
    begin buffer[tmp_ptr] ← sv_buffer[tmp_ptr]; incr(tmp_ptr);
    end;
exit: end;
```

212. A `reverse` command has one argument, a single *built_in* or *wiz_defined* function name between braces. Upper/lower cases are considered to be the same—all upper-case letters are converted to lower case. Also, we must make sure we've already seen a `read` command.

This module reads a *left_brace*, a single function to be iterated in reverse, and a *right_brace*.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_reverse_command;
  label exit;
  begin if (¬read_seen) then bst_err('Illegal,␣reverse␣command␣before␣read␣command');
  eat_bst_white_and_eof_check('reverse'); bst_get_and_check_left_brace('reverse');
  eat_bst_white_and_eof_check('reverse'); bst_identifier_scan('reverse');
  ⟨ Check the reverse-command argument token 213 ⟩;
  eat_bst_white_and_eof_check('reverse'); bst_get_and_check_right_brace('reverse');
  ⟨ Perform a reverse command 298 ⟩;
exit: end;
```

213. Before iterating the function in reverse, we must make sure it's a legal one. It must exist and be *built_in* or *wiz_defined*.

```
⟨ Check the reverse-command argument token 213 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln('␣is␣a␣to␣be␣iterated␣in␣reverse␣function');
  ecart
  if (bad_argument_token) then return;
  end
```

This code is used in section 212.

214. The `sort` command has no arguments so there's no more parsing to do, but we must make sure we've already seen a `read` command.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_sort_command;
  label exit;
  begin if (¬read_seen) then bst_err('Illegal,␣sort␣command␣before␣read␣command');
  ⟨ Perform a sort command 299 ⟩;
exit: end;
```

215. A `strings` command has one argument, a list of function names between braces (the names are separated by one or more *white_space* characters). Upper/lower cases are considered to be the same for function names in these lists—all upper-case letters are converted to lower case. Each name in this list specifies a *str_global_var*. There may be several `strings` commands in the `.bst` file.

This module reads a *left_brace*, a list of *str_global_var*s, and a *right_brace*.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
procedure bst_strings_command;
  label exit;
  begin eat_bst_white_and_eof_check('strings'); bst_get_and_check_left_brace('strings');
  eat_bst_white_and_eof_check('strings');
  while (scan_char ≠ right_brace) do
    begin bst_identifier_scan('strings'); ⟨ Insert a str_global_var into the hash table 216 ⟩;
    eat_bst_white_and_eof_check('strings');
    end;
  incr(buf_ptr2); { skip over the right_brace }
exit: end;
```

216. Here we insert the just found *str_global_var* name into the hash table, record it as a *str_global_var*, set its pointer into *global_strs*, and initialize its value there to the null string.

```
define end_of_string = invalid_code  { this illegal ASCII_code ends a string }
⟨ Insert a str_global_var into the hash table 216 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln(´␣is␣a␣string␣global-variable´);
  ecart
  lower_case(buffer, buf_ptr1, token_len);  { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, do_insert);
  check_for_already_seen_function(fn_loc); fn_type[fn_loc] ← str_global_var;
  fn_info[fn_loc] ← num_glb_strs;  { pointer into global_strs }
  if (num_glb_strs = max_glob_strs) then
    overflow(´number␣of␣string␣global-variables␣´, max_glob_strs);
  incr(num_glb_strs);
  end
```

This code is used in section 215.

217. That's it for processing `.bst` commands, except for finishing the procedural gymnastics. Note that this must topologically follow the stuff for `.bib` reading, because that's done by the `.bst`'s `read` command.

```
⟨ Procedures and functions for the reading and processing of input files 100 ⟩ +≡
  ⟨ Scan for and process a .bst command 154 ⟩
```

218. Reading the database file(s). This section reads the `.bib` file(s), each of which consists of a sequence of entries (perhaps with a few `.bib` commands thrown in, as explained later). Each entry consists of an *at_sign*, an entry type, and, between braces or parentheses and separated by *comma*s, a database key and a list of fields. Each field consists of a field name, an *equals_sign*, and nonempty list of field tokens separated by *concat_char*s. Each field token is either a nonnegative number, a macro name (like 'jan'), or a brace-balanced string delimited by either *double_quote*s or braces. Finally, case differences are ignored for all but delimited strings and database keys, and *white_space* characters and ends-of-line may appear in all reasonable places (i.e., anywhere except within entry types, database keys, field names, and macro names); furthermore, comments may appear anywhere between entries (or before the first or after the last) as long as they contain no *at_sign*s.

219. These global variables are used while reading the `.bib` file(s). The elements of *type_list*, which indicate an entry's type (book, article, etc.), point either to a *hash_loc* or are one of two special markers: *empty*, from which *hash_base* = *empty* + 1 was defined, means we haven't yet encountered the `.bib` entry corresponding to this cite key; and *undefined* means we've encountered it but it had an unknown entry type. Thus the array *type_list* is of type *hash_ptr2*, also defined earlier. An element of the boolean array *entry_exists* whose corresponding entry in *cite_list* gets overwritten (which happens only when *all_entries* is *true*) indicates whether we've encountered that entry of *cite_list* while reading the `.bib` file(s); this information is unused for entries that aren't (or more precisely, that have no chance of being) overwritten. When we're reading the database file, the array *cite_info* contains auxiliary information for *cite_list*. Later, *cite_info* will become *sorted_cites*, and this dual role imposes the (not-very-imposing) restriction *max_strings* ≥ *max_cites*.

define *undefined* = *hash_max* + 1 { a special marker used for *type_list* }

⟨ Globals in the outer block 16 ⟩ +≡
bib_line_num: *integer*; { line number of the `.bib` file }
entry_type_loc: *hash_loc*; { the hash-table location of an entry type }
type_list: **packed array** [*cite_number*] **of** *hash_ptr2*;
type_exists: *boolean*; { *true* if this entry type is `.bst`-defined }
entry_exists: **packed array** [*cite_number*] **of** *boolean*;
store_entry: *boolean*; { *true* if we're to store info for this entry }
field_name_loc: *hash_loc*; { the hash-table location of a field name }
field_val_loc: *hash_loc*; { the hash-table location of a field value }
store_field: *boolean*; { *true* if we're to store info for this field }
store_token: *boolean*; { *true* if we're to store this macro token }
right_outer_delim: *ASCII_code*; { either a *right_brace* or a *right_paren* }
right_str_delim: *ASCII_code*; { either a *right_brace* or a *double_quote* }
at_bib_command: *boolean*; { *true* for a command, false for an entry }
cur_macro_loc: *hash_loc*; { *macro_loc* for a `string` being defined }
cite_info: **packed array** [*cite_number*] **of** *str_number*; { extra *cite_list* info }
cite_hash_found: *boolean*; { set to a previous *hash_found* value }
preamble_ptr: *bib_number*; { pointer into the *s_preamble* array }
num_preamble_strings: *bib_number*; { counts the *s_preamble* strings }

220. This little procedure exists because it's used by at least two other procedures and thus saves some space.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure *bib_ln_num_print*;
begin *print*(´`--line␣`´, *bib_line_num* : 0, ´`␣of␣file␣`´); *print_bib_name*;
end;

221. When there's a serious error parsing a `.bib` file, we flush everything up to the beginning of the next entry.

```
define bib_err(#) ≡
        begin  { serious error during .bib parsing }
        print(#); bib_err_print; return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bib_err_print;
  begin print('-'); bib_ln_num_print; print_bad_input_line;  { this call does the mark_error }
  print_skipping_whatever_remains;
  if (at_bib_command) then print_ln('command')
  else print_ln('entry');
  end;
```

222. When there's a harmless error parsing a `.bib` file, we just give a warning message. This is always called after other stuff has been printed out.

```
define bib_warn(#) ≡
        begin  { non-serious error during .bst parsing }
        print(#); bib_warn_print;
        end
define bib_warn_newline(#) ≡
        begin  { same as above but with a newline }
        print_ln(#); bib_warn_print;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bib_warn_print;
  begin bib_ln_num_print; mark_warning;
  end;
```

223. For all *num_bib_files* database files, we keep reading and processing `.bib` entries until none left.

⟨ Read the `.bib` file(s) 223 ⟩ ≡

```
  begin ⟨ Final initialization for .bib processing 224 ⟩;
  read_performed ← true; bib_ptr ← 0;
  while (bib_ptr < num_bib_files) do
    begin print('Database␣file␣#', bib_ptr + 1 : 0, ':␣'); print_bib_name;
    bib_line_num ← 0;  { initialize to get the first input line }
    buf_ptr2 ← last;
    while (¬eof(cur_bib_file)) do get_bib_command_or_entry_and_process;
    a_close(cur_bib_file); incr(bib_ptr);
    end;
  reading_completed ← true;
  trace pr_ln('Finished␣reading␣the␣database␣file(s)');
  ecart
  ⟨ Final initialization for processing the entries 276 ⟩;
  read_completed ← true;
  end
```

This code is used in section 211.

224. We need to initialize the *field_info* array, and also various things associated with the *cite_list* array (but not *cite_list* itself).

⟨ Final initialization for `.bib` processing 224 ⟩ ≡
```
begin ⟨ Initialize the field_info 225 ⟩;
⟨ Initialize things for the cite_list 227 ⟩;
end
```
This code is used in section 223.

225. This module initializes all fields of all entries to *missing*, the value to which all fields are initialized.

⟨ Initialize the *field_info* 225 ⟩ ≡
```
begin check_field_overflow(num_fields * num_cites); field_ptr ← 0;
while (field_ptr < max_fields) do
  begin field_info[field_ptr] ← missing; incr(field_ptr);
  end;
end
```
This code is used in section 224.

226. Complain if somebody's got a field fetish.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
```
procedure check_field_overflow(total_fields : integer);
  begin if (total_fields > max_fields) then
    begin print_ln(total_fields : 0, ´␣fields:´); overflow(´total␣number␣of␣fields␣´, max_fields);
    end;
  end;
```

227. We must initialize the *type_list* array so that we can detect duplicate (or missing) entries for cite keys on *cite_list*. Also, when we're to include the entire database, we use the array *entry_exists* to detect those missing entries whose *cite_list* info will (or to be more precise, might) be overwritten; and we use the array *cite_info* to save the part of *cite_list* that will (might) be overwritten. We also use *cite_info* for counting cross references when it's appropriate—when an entry isn't otherwise to be included on *cite_list* (that is, the entry isn't `\cited` or `\nocited`). Such an entry is included on the final *cite_list* if it's cross referenced at least *min_crossrefs* times.

```
⟨ Initialize things for the cite_list 227 ⟩ ≡
  begin cite_ptr ← 0;
  while (cite_ptr < max_cites) do
    begin type_list[cite_ptr] ← empty;
    cite_info[cite_ptr] ← any_value;  { to appeas PASCAL's boolean evaluation }
    incr(cite_ptr);
    end;
  old_num_cites ← num_cites;
  if (all_entries) then
    begin cite_ptr ← all_marker;
    while (cite_ptr < old_num_cites) do
      begin cite_info[cite_ptr] ← cite_list[cite_ptr]; entry_exists[cite_ptr] ← false; incr(cite_ptr);
      end;
    cite_ptr ← all_marker;  { we insert the "other" entries here }
    end
  else begin cite_ptr ← num_cites;  { we insert the cross-referenced entries here }
    all_marker ← any_value;  { to appease PASCAL's boolean evaluation }
    end;
  end
```

This code is used in section 224.

228. Before we actually start the code for reading a database file, we must define this `.bib`-specific scanning function. It skips over *white_space* characters until hitting a nonwhite character or the end of the file, respectively returning *true* or *false*. It also updates *bib_line_num*, the line counter.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function eat_bib_white_space: boolean;
  label exit;
  begin while (¬scan_white_space) do  { no characters left; read another line }
    begin if (¬input_ln(cur_bib_file)) then  { end-of-file; return false }
      begin eat_bib_white_space ← false; return;
      end;
    incr(bib_line_num); buf_ptr2 ← 0;
    end;
  eat_bib_white_space ← true;
exit: end;
```

229. It's often illegal to end a `.bib` command in certain places, and this is where we come to check.

```
define eat_bib_white_and_eof_check ≡
        begin if (¬eat_bib_white_space) then
          begin eat_bib_print; return;
          end;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure eat_bib_print;
  label exit; { so the call to bib_err works }
  begin bib_err(´Illegal␣end␣of␣database␣file´);
exit: end;
```

230. And here are a bunch of error-message macros, each called more than once, that thus save space as implemented. This one is for when one of two possible characters is expected while scanning.

```
define bib_one_of_two_expected_err(#) ≡
        begin bib_one_of_two_print(#); return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bib_one_of_two_print(char1 , char2 : ASCII_code);
  label exit; { so the call to bib_err works }
  begin bib_err(´I␣was␣expecting␣a␣`´, xchr[char1], ´´´␣or␣a␣`´, xchr[char2], ´´´´);
exit: end;
```

231. This one's for an expected *equals_sign*.

```
define bib_equals_sign_expected_err ≡
        begin bib_equals_sign_print; return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bib_equals_sign_print;
  label exit; { so the call to bib_err works }
  begin bib_err(´I␣was␣expecting␣an␣"´, xchr[equals_sign], ´"´);
exit: end;
```

232. This complains about unbalanced braces.

```
define bib_unbalanced_braces_err ≡
        begin bib_unbalanced_braces_print; return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bib_unbalanced_braces_print;
  label exit; { so the call to bib_err works }
  begin bib_err(´Unbalanced␣braces´);
exit: end;
```

233. And this one about an overly exuberant field.

```
define bib_field_too_long_err ≡
        begin bib_field_too_long_print; return;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bib_field_too_long_print;
  label exit;  { so the call to bib_err works }
  begin bib_err('Your␣field␣is␣more␣than␣', buf_size : 0, '␣characters');
exit: end;
```

234. This one is just a warning, not an error. It's for when something isn't (or might not be) quite right with a macro name.

```
define macro_name_warning(#) ≡
        begin macro_warn_print; bib_warn_newline(#);
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure macro_warn_print;
  begin print('Warning--string␣name␣"'); print_token; print('"␣is␣');
  end;
```

235. This macro is used to scan all `.bib` identifiers. The argument tells what was happening at the time. The associated procedure simply prints an error message.

```
define bib_identifier_scan_check(#) ≡
        begin if ((scan_result = white_adjacent) ∨ (scan_result = specified_char_adjacent)) then
          do_nothing
        else begin bib_id_print; bib_err(#);
          end;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bib_id_print;
  begin if (scan_result = id_null) then print('You''re␣missing␣')
  else if (scan_result = other_char_adjacent) then
      print('"', xchr[scan_char], '"␣immediately␣follows␣')
    else id_scanning_confusion;
  end;
```

236. This module either reads a database entry, whose three main components are an entry type, a database key, and a list of fields, or it reads a `.bib` command, whose structure is command dependent and explained later.

```
define cite_already_set = 22  { this gets around PASCAL limitations }
define first_time_entry = 26  { for checking for repeated database entries }
```

```
⟨ Scan for and process a .bib command or database entry 236 ⟩ ≡
procedure get_bib_command_or_entry_and_process;
  label cite_already_set, first_time_entry, loop_exit, exit;
  begin at_bib_command ← false;
  ⟨ Skip to the next database entry or .bib command 237 ⟩;
  ⟨ Scan the entry type or scan and process the .bib command 238 ⟩;
  eat_bib_white_and_eof_check; ⟨ Scan the entry's database key 266 ⟩;
  eat_bib_white_and_eof_check; ⟨ Scan the entry's list of fields 274 ⟩;
exit: end;
```

This code is used in section 210.

237. This module skips over everything until hitting an *at_sign* or the end of the file. It also updates *bib_line_num*, the line counter.

```
⟨ Skip to the next database entry or .bib command 237 ⟩ ≡
  while (¬scan1(at_sign)) do  { no at_sign; get next line }
    begin if (¬input_ln(cur_bib_file)) then  { end-of-file }
      return;
    incr(bib_line_num); buf_ptr2 ← 0;
    end
```

This code is used in section 236.

238. This module reads an *at_sign* and an entry type (like 'book' or 'article') or a `.bib` command. If it's an entry type, it must be defined in the `.bst` file if this entry is to be included in the reference list.

```
⟨ Scan the entry type or scan and process the .bib command 238 ⟩ ≡
  begin if (scan_char ≠ at_sign) then confusion('An␣"', xchr[at_sign], '"␣disappeared');
  incr(buf_ptr2);  { skip over the at_sign }
  eat_bib_white_and_eof_check; scan_identifier(left_brace, left_paren, left_paren);
  bib_identifier_scan_check('an␣entry␣type');
  trace trace_pr_token; trace_pr_ln('␣is␣an␣entry␣type␣or␣a␣database-file␣command');
  ecart
  lower_case(buffer, buf_ptr1, token_len);  { ignore case differences }
  command_num ← ilk_info[str_lookup(buffer, buf_ptr1, token_len, bib_command_ilk, dont_insert)];
  if (hash_found) then ⟨ Process a .bib command 239 ⟩
  else begin  { process an entry type }
    entry_type_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, dont_insert);
    if ((¬hash_found) ∨ (fn_type[entry_type_loc] ≠ wiz_defined)) then
      type_exists ← false  { no such entry type defined in the .bst file }
    else type_exists ← true;
    end;
  end
```

This code is used in section 236.

239. Here we determine which `.bib` command we're about to process, then go to it.

⟨ Process a `.bib` command 239 ⟩ ≡
```
begin at_bib_command ← true;
case (command_num) of
n_bib_comment: ⟨ Process a comment command 241 ⟩;
n_bib_preamble: ⟨ Process a preamble command 242 ⟩;
n_bib_string: ⟨ Process a string command 243 ⟩;
othercases bib_cmd_confusion
endcases;
end
```
This code is used in section 238.

240. Here's another bug.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
```
procedure bib_cmd_confusion;
  begin confusion('Unknown␣database-file␣command');
  end;
```

241. The `comment` command is implemented for SCRIBE compatibility. It's not really needed because BIBTEX treats (flushes) everything not within an entry as a comment anyway.

⟨ Process a `comment` command 241 ⟩ ≡
```
begin return;  { flush comments }
end
```
This code is used in section 239.

242. The `preamble` command lets a user have TEX stuff inserted (by the standard styles, at least) directly into the `.bbl` file. It is intended primarily for allowing TEX macro definitions used within the bibliography entries (for better sorting, for example). One `preamble` command per `.bib` file should suffice.

A `preamble` command has either braces or parentheses as outer delimiters. Inside is the preamble string, which has the same syntax as a field value: a nonempty list of field tokens separated by *concat_char*s. There are three types of field tokens—nonnegative numbers, macro names, and delimited strings.

This module does all the scanning (that's not subcontracted), but the `.bib`-specific scanning function *scan_and_store_the_field_value_and_eat_white* actually stores the value.

⟨ Process a `preamble` command 242 ⟩ ≡
```
begin if (preamble_ptr = max_bib_files) then
    bib_err('You''ve␣exceeded␣', max_bib_files : 0, '␣preamble␣commands');
  eat_bib_white_and_eof_check;
  if (scan_char = left_brace) then right_outer_delim ← right_brace
  else if (scan_char = left_paren) then right_outer_delim ← right_paren
    else bib_one_of_two_expected_err(left_brace, left_paren);
  incr(buf_ptr2);  { skip over the left-delimiter }
  eat_bib_white_and_eof_check; store_field ← true;
  if (¬scan_and_store_the_field_value_and_eat_white) then return;
  if (scan_char ≠ right_outer_delim) then
    bib_err('Missing␣"', xchr[right_outer_delim], '"␣in␣preamble␣command');
  incr(buf_ptr2);  { skip over the right_outer_delim }
  return;
  end
```
This code is used in section 239.

243. The `string` command is implemented both for SCRIBE compatibility and for allowing a user: to override a `.bst`-file `macro` command, to define one that the `.bst` file doesn't, or to engage in good, wholesome, typing laziness.

The `string` command does mostly the same thing as the `.bst`-file's `macro` command (but the syntax is different and the `string` command compresses *white_space*). In fact, later in this program, the term "macro" refers to either a `.bst` "macro" or a `.bib` "string" (when it's clear from the context that it's not a `WEB` macro).

A `string` command has either braces or parentheses as outer delimiters. Inside is the string's name (it must be a legal identifier, and case differences are ignored—all upper-case letters are converted to lower case), then an *equals_sign*, and the string's definition, which has the same syntax as a field value: a nonempty list of field tokens separated by *concat_char*s. There are three types of field tokens—nonnegative numbers, macro names, and delimited strings.

```
⟨ Process a string command 243 ⟩ ≡
  begin eat_bib_white_and_eof_check; ⟨ Scan the string's name 244 ⟩;
  eat_bib_white_and_eof_check; ⟨ Scan the string's definition field 246 ⟩;
  return;
  end
```

This code is used in section 239.

244. This module reads a left outer-delimiter and a string name.

```
⟨ Scan the string's name 244 ⟩ ≡
  begin if (scan_char = left_brace) then right_outer_delim ← right_brace
  else if (scan_char = left_paren) then right_outer_delim ← right_paren
    else bib_one_of_two_expected_err(left_brace, left_paren);
  incr(buf_ptr2); { skip over the left-delimiter }
  eat_bib_white_and_eof_check; scan_identifier(equals_sign, equals_sign, equals_sign);
  bib_identifier_scan_check('a␣string␣name'); ⟨ Store the string's name 245 ⟩;
  end
```

This code is used in section 243.

245. This module marks this string as *macro_ilk*; the commented-out code will give a warning message when overwriting a previously defined macro.

```
⟨ Store the string's name 245 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln('␣is␣a␣database-defined␣macro');
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  cur_macro_loc ← str_lookup(buffer, buf_ptr1, token_len, macro_ilk, do_insert);
  ilk_info[cur_macro_loc] ← hash_text[cur_macro_loc]; { default in case of error }
  @{
    if (hash_found) then { already seen macro }
      macro_name_warning('having␣its␣definition␣overwritten');
  @}
  end
```

This code is used in section 244.

246. This module skips over the *equals_sign*, reads and stores the list of field tokens that defines this macro (compressing *white_space*), and reads a *right_outer_delim*.

```
⟨ Scan the string's definition field 246 ⟩ ≡
  begin if (scan_char ≠ equals_sign) then bib_equals_sign_expected_err;
  incr(buf_ptr2); { skip over the equals_sign }
  eat_bib_white_and_eof_check; store_field ← true;
  if (¬scan_and_store_the_field_value_and_eat_white) then return;
  if (scan_char ≠ right_outer_delim) then
    bib_err('Missing␣"', xchr[right_outer_delim], '"␣in␣string␣command');
  incr(buf_ptr2); { skip over the right_outer_delim }
  end
```

This code is used in section 243.

247. The variables for the function *scan_and_store_the_field_value_and_eat_white* must be global since the functions it calls use them too. The alias kludge helps make the stack space not overflow on some machines.

```
  define field_vl_str ≡ ex_buf { aliases, used "only" for this function }
  define field_end ≡ ex_buf_ptr { the end marker for the field-value string }
  define field_start ≡ ex_buf_xptr { and the start marker }
⟨ Globals in the outer block 16 ⟩ +≡
bib_brace_level: integer; { brace nesting depth (excluding str_delims) }
```

248. Since the function *scan_and_store_the_field_value_and_eat_white* calls several other yet-to-be-described functions (one directly and two indirectly), we must perform some topological gymnastics.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
  ⟨ The scanning function compress_bib_white 252 ⟩
  ⟨ The scanning function scan_balanced_braces 253 ⟩
  ⟨ The scanning function scan_a_field_token_and_eat_white 250 ⟩
```

249. This function scans the list of field tokens that define the field value string. If *store_field* is *true* it accumulates (indirectly) in *field_vl_str* the concatenation of all the field tokens, compressing nonnull *white_space* to a single *space* and, if the field value is for a field (rather than a string definition), removing any leading or trailing *white_space*; when it's finished it puts the string into the hash table. It returns *false* if there was a serious syntax error.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
function scan_and_store_the_field_value_and_eat_white: boolean;
  label exit;
  begin scan_and_store_the_field_value_and_eat_white ← false; { now it's easy to exit if necessary }
  field_end ← 0;
  if (¬scan_a_field_token_and_eat_white) then return;
  while (scan_char = concat_char) do { scan remaining field tokens }
    begin incr(buf_ptr2); { skip over the concat_char }
    eat_bib_white_and_eof_check;
    if (¬scan_a_field_token_and_eat_white) then return;
    end;
  if (store_field) then ⟨ Store the field value string 261 ⟩;
  scan_and_store_the_field_value_and_eat_white ← true;
exit: end;
```

250. Each field token is either a nonnegative number, a macro name (like 'jan'), or a brace-balanced string delimited by either *double_quote*s or braces. Thus there are four possibilities for the first character of the field token: If it's a *left_brace* or a *double_quote*, the token (with balanced braces, up to the matching *right_str_delim*) is a string; if it's *numeric*, the token is a number; if it's anything else, the token is a macro name (and should thus have been defined by either the `.bst`-file's `macro` command or the `.bib`-file's `string` command). This function returns *false* if there was a serious syntax error.

```
⟨ The scanning function scan_a_field_token_and_eat_white 250 ⟩ ≡
function scan_a_field_token_and_eat_white: boolean;
  label exit;
  begin scan_a_field_token_and_eat_white ← false;  { now it's easy to exit if necessary }
  case (scan_char) of
  left_brace: begin right_str_delim ← right_brace;
    if (¬scan_balanced_braces) then return;
    end;
  double_quote: begin right_str_delim ← double_quote;
    if (¬scan_balanced_braces) then return;
    end;
  "0", "1", "2", "3", "4", "5", "6", "7", "8", "9": ⟨ Scan a number 258 ⟩;
  othercases ⟨ Scan a macro name 259 ⟩
  endcases; eat_bib_white_and_eof_check; scan_a_field_token_and_eat_white ← true;
exit: end;
```

This code is used in section 248.

251. Now we come to the stuff that actually accumulates the field value to be stored. This module copies a character into *field_vl_str* if it will fit; since it's so low level, it's implemented as a macro.

```
define copy_char(#) ≡
        begin if (field_end = buf_size) then bib_field_too_long_err
        else begin field_vl_str[field_end] ← #; incr(field_end);
          end;
        end
```

252. The `.bib`-specific scanning function *compress_bib_white* skips over *white_space* characters within a string until hitting a nonwhite character; in fact, it does everything *eat_bib_white_space* does, but it also adds a *space* to *field_vl_str*. This function is never called if there are no *white_space* characters (or ends-of-line) to be scanned (though the associated macro might be). The function returns *false* if there is a serious syntax error.

```
define check_for_and_compress_bib_white_space ≡
        begin if ((lex_class[scan_char] = white_space) ∨ (buf_ptr2 = last)) then
          if (¬compress_bib_white) then return;
        end
```

⟨ The scanning function *compress_bib_white* 252 ⟩ ≡

```
function compress_bib_white: boolean;
  label exit;
  begin compress_bib_white ← false;  { now it's easy to exit if necessary }
  copy_char(space);
  while (¬scan_white_space) do  { no characters left; read another line }
    begin if (¬input_ln(cur_bib_file)) then  { end-of-file; complain }
      begin eat_bib_print; return;
      end;
    incr(bib_line_num); buf_ptr2 ← 0;
    end;
  compress_bib_white ← true;
exit: end;
```

This code is used in section 248.

253. This `.bib`-specific function scans a string with balanced braces, stopping just past the matching *right_str_delim*. How much work it does depends on whether *store_field* = *true*. It returns *false* if there was a serious syntax error.

⟨ The scanning function *scan_balanced_braces* 253 ⟩ ≡

```
function scan_balanced_braces: boolean;
  label loop_exit, exit;
  begin scan_balanced_braces ← false;  { now it's easy to exit if necessary }
  incr(buf_ptr2);  { skip over the left-delimiter }
  check_for_and_compress_bib_white_space;
  if (field_end > 1) then
    if (field_vl_str[field_end − 1] = space) then
      if (field_vl_str[field_end − 2] = space) then  { remove wrongly added space }
        decr(field_end);
  bib_brace_level ← 0;  { and we're at a nonwhite_space character }
  if (store_field) then ⟨ Do a full brace-balanced scan 256 ⟩
  else ⟨ Do a quick brace-balanced scan 254 ⟩;
  incr(buf_ptr2);  { skip over the right_str_delim }
  scan_balanced_braces ← true;
exit: end;
```

This code is used in section 248.

254. This module scans over a brace-balanced string without keeping track of anything but the brace level. It starts with *bib_brace_level* = 0 and at a non*white_space* character.

```
⟨ Do a quick brace-balanced scan 254 ⟩ ≡
  begin while (scan_char ≠ right_str_delim) do  { we're at bib_brace_level = 0 }
    if (scan_char = left_brace) then
      begin incr(bib_brace_level); incr(buf_ptr2);  { skip over the left_brace }
      eat_bib_white_and_eof_check;
      while (bib_brace_level > 0) do ⟨ Do a quick scan with bib_brace_level > 0 255 ⟩;
      end
    else if (scan_char = right_brace) then bib_unbalanced_braces_err
      else begin incr(buf_ptr2);  { skip over some other character }
        if (¬scan3(right_str_delim, left_brace, right_brace)) then eat_bib_white_and_eof_check;
        end
  end
```

This code is used in section 253.

255. This module does the same as above but, because *bib_brace_level* > 0, it doesn't have to look for a *right_str_delim*.

```
⟨ Do a quick scan with bib_brace_level > 0 255 ⟩ ≡
  begin   { top part of the while loop—we're always at a nonwhite character }
  if (scan_char = right_brace) then
    begin decr(bib_brace_level); incr(buf_ptr2);  { skip over the right_brace }
    eat_bib_white_and_eof_check;
    end
  else if (scan_char = left_brace) then
      begin incr(bib_brace_level); incr(buf_ptr2);  { skip over the left_brace }
      eat_bib_white_and_eof_check;
      end
    else begin incr(buf_ptr2);  { skip over some other character }
      if (¬scan2(right_brace, left_brace)) then eat_bib_white_and_eof_check;
      end
  end
```

This code is used in section 254.

256. This module scans over a brace-balanced string, compressing multiple *white_space* characters into a single *space*. It starts with *bib_brace_level* = 0 and starts at a non*white_space* character.

```
⟨ Do a full brace-balanced scan 256 ⟩ ≡
  begin while (scan_char ≠ right_str_delim) do
    case (scan_char) of
    left_brace: begin incr(bib_brace_level); copy_char(left_brace);
      incr(buf_ptr2);  { skip over the left_brace }
      check_for_and_compress_bib_white_space;
      ⟨ Do a full scan with bib_brace_level > 0 257 ⟩;
      end;
    right_brace: bib_unbalanced_braces_err;
    othercases begin copy_char(scan_char); incr(buf_ptr2);  { skip over some other character }
      check_for_and_compress_bib_white_space;
      end
    endcases;
  end
```

This code is used in section 253.

257. This module is similar to the last but starts with *bib_brace_level* > 0 (and, like the last, it starts at a non*white_space* character).

```
⟨ Do a full scan with bib_brace_level > 0 257 ⟩ ≡
  begin loop
    case (scan_char) of
    right_brace: begin decr(bib_brace_level); copy_char(right_brace);
      incr(buf_ptr2); { skip over the right_brace }
      check_for_and_compress_bib_white_space;
      if (bib_brace_level = 0) then goto loop_exit;
      end;
    left_brace: begin incr(bib_brace_level); copy_char(left_brace);
      incr(buf_ptr2); { skip over the left_brace }
      check_for_and_compress_bib_white_space;
      end;
    othercases begin copy_char(scan_char); incr(buf_ptr2); { skip over some other character }
      check_for_and_compress_bib_white_space;
      end
    endcases;
loop_exit: end
```

This code is used in section 256.

258. This module scans a nonnegative number and copies it to *field_vl_str* if it's to store the field.

```
⟨ Scan a number 258 ⟩ ≡
  begin if (¬scan_nonneg_integer) then confusion('A␣digit␣disappeared');
  if (store_field) then
    begin tmp_ptr ← buf_ptr1;
    while (tmp_ptr < buf_ptr2) do
      begin copy_char(buffer[tmp_ptr]); incr(tmp_ptr);
      end;
    end;
  end
```

This code is used in section 250.

259. This module scans a macro name and copies its string to *field_vl_str* if it's to store the field, complaining if the macro is recursive or undefined.

```
⟨ Scan a macro name 259 ⟩ ≡
  begin scan_identifier(comma, right_outer_delim, concat_char);
  bib_identifier_scan_check('a␣field␣part');
  if (store_field) then
    begin lower_case(buffer, buf_ptr1, token_len);  { ignore case differences }
    macro_name_loc ← str_lookup(buffer, buf_ptr1, token_len, macro_ilk, dont_insert); store_token ← true;
    if (at_bib_command) then
      if (command_num = n_bib_string) then
        if (macro_name_loc = cur_macro_loc) then
          begin store_token ← false; macro_name_warning('used␣in␣its␣own␣definition');
          end;
    if (¬hash_found) then
      begin store_token ← false; macro_name_warning('undefined');
      end;
    if (store_token) then ⟨ Copy the macro string to field_vl_str 260 ⟩;
    end;
  end
```

This code is used in section 250.

260. The macro definition may have *white_space* that needs compressing, because it may have been defined in the `.bst` file.

```
⟨ Copy the macro string to field_vl_str 260 ⟩ ≡
  begin tmp_ptr ← str_start[ilk_info[macro_name_loc]];
  tmp_end_ptr ← str_start[ilk_info[macro_name_loc] + 1];
  if (field_end = 0) then
    if ((lex_class[str_pool[tmp_ptr]] = white_space) ∧ (tmp_ptr < tmp_end_ptr)) then
      begin  { compress leading white_space of first nonnull token }
      copy_char(space); incr(tmp_ptr);
      while ((lex_class[str_pool[tmp_ptr]] = white_space) ∧ (tmp_ptr < tmp_end_ptr)) do incr(tmp_ptr);
      end;  { the next remaining character is non white_space }
  while (tmp_ptr < tmp_end_ptr) do
    begin if (lex_class[str_pool[tmp_ptr]] ≠ white_space) then copy_char(str_pool[tmp_ptr])
    else if (field_vl_str[field_end − 1] ≠ space) then copy_char(space);
    incr(tmp_ptr);
    end;
  end
```

This code is used in section 259.

261. Now it's time to store the field value in the hash table, and store an appropriate pointer to it (depending on whether it's for a database entry or command). But first, if necessary, we remove a trailing *space* and a leading *space* if these exist. (Hey, if we had some ham we could make ham-and-eggs if we had some eggs.)

```
⟨ Store the field value string 261 ⟩ ≡
  begin if (¬at_bib_command) then   { chop trailing space for a field }
    if (field_end > 0) then
      if (field_vl_str[field_end − 1] = space) then decr(field_end);
  if ((¬at_bib_command) ∧ (field_vl_str[0] = space) ∧ (field_end > 0)) then   { chop leading space for a field }
    field_start ← 1
  else field_start ← 0;
  field_val_loc ← str_lookup(field_vl_str, field_start, field_end − field_start, text_ilk, do_insert);
  fn_type[field_val_loc] ← str_literal;   { set the fn_class }
  trace trace_pr(´"´); trace_pr_pool_str(hash_text[field_val_loc]); trace_pr_ln(´"␣is␣a␣field␣value´);
  ecart
  if (at_bib_command) then   { for a preamble or string command }
    ⟨ Store the field value for a command 262 ⟩
  else   { for a database entry }
  ⟨ Store the field value for a database entry 263 ⟩;
  end
```

This code is used in section 249.

262. Here's where we store the goods when we're dealing with a command rather than an entry.

```
⟨ Store the field value for a command 262 ⟩ ≡
  begin case (command_num) of
  n_bib_preamble: begin s_preamble[preamble_ptr] ← hash_text[field_val_loc]; incr(preamble_ptr);
    end;
  n_bib_string: ilk_info[cur_macro_loc] ← hash_text[field_val_loc];
  othercases bib_cmd_confusion
  endcases;
  end
```

This code is used in section 261.

263. And here, an entry.

```
⟨ Store the field value for a database entry 263 ⟩ ≡
  begin field_ptr ← entry_cite_ptr ∗ num_fields + fn_info[field_name_loc];
  if (field_info[field_ptr] ≠ missing) then
    begin print(´Warning--I´´m␣ignoring␣´); print_pool_str(cite_list[entry_cite_ptr]);
    print(´´´s␣extra␣"´); print_pool_str(hash_text[field_name_loc]); bib_warn_newline(´"␣field´);
    end
  else begin   { the field was empty, store its new value }
    field_info[field_ptr] ← hash_text[field_val_loc];
    if ((fn_info[field_name_loc] = crossref_num) ∧ (¬all_entries)) then
      ⟨ Add or update a cross reference on cite_list if necessary 264 ⟩;
    end;
  end
```

This code is used in section 261.

264. If the cross-referenced entry isn't already on *cite_list* we add it (at least temporarily); if it is already on *cite_list* we update the cross-reference count, if necessary. Note that *all_entries* is *false* here. The alias kludge helps make the stack space not overflow on some machines.

```
define extra_buf ≡ out_buf  { an alias, used only in this module }
⟨ Add or update a cross reference on cite_list if necessary 264 ⟩ ≡
  begin tmp_ptr ← field_start;
  while (tmp_ptr < field_end) do
    begin extra_buf[tmp_ptr] ← field_vl_str[tmp_ptr]; incr(tmp_ptr);
    end;
  lower_case(extra_buf, field_start, field_end − field_start);  { convert to 'canonical' form }
  lc_cite_loc ← str_lookup(extra_buf, field_start, field_end − field_start, lc_cite_ilk, do_insert);
  if (hash_found) then
    begin cite_loc ← ilk_info[lc_cite_loc];  { even if there's a case mismatch }
    if (ilk_info[cite_loc] ≥ old_num_cites) then  { a previous crossref }
      incr(cite_info[ilk_info[cite_loc]]);
    end
  else begin  { it's a new crossref }
    cite_loc ← str_lookup(field_vl_str, field_start, field_end − field_start, cite_ilk, do_insert);
    if (hash_found) then hash_cite_confusion;
    add_database_cite(cite_ptr);  { this increments cite_ptr }
    cite_info[ilk_info[cite_loc]] ← 1;  { the first cross-ref for this cite key }
    end;
  end
```

This code is used in section 263.

265. This procedure adds (or restores) to *cite_list* a cite key; it is called only when *all_entries* is *true* or when adding cross references, and it assumes that *cite_loc* and *lc_cite_loc* are set. It also increments its argument.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure add_database_cite(var new_cite : cite_number);
  begin check_cite_overflow(new_cite);  { make sure this cite will fit }
  check_field_overflow(num_fields * new_cite); cite_list[new_cite] ← hash_text[cite_loc];
  ilk_info[cite_loc] ← new_cite; ilk_info[lc_cite_loc] ← cite_loc; incr(new_cite);
  end;
```

266. And now, back to processing an entry (rather than a command). This module reads a left outer-delimiter and a database key.

```
⟨ Scan the entry's database key 266 ⟩ ≡
  begin if (scan_char = left_brace) then right_outer_delim ← right_brace
  else if (scan_char = left_paren) then right_outer_delim ← right_paren
    else bib_one_of_two_expected_err(left_brace, left_paren);
  incr(buf_ptr2); { skip over the left-delimiter }
  eat_bib_white_and_eof_check;
  if (right_outer_delim = right_paren) then { to allow it in a database key }
    begin if (scan1_white(comma)) then { ok if database key ends line }
      do_nothing;
    end
  else if (scan2_white(comma, right_brace)) then { right_brace = right_outer_delim }
      do_nothing;
  ⟨ Check for a database key of interest 267 ⟩;
  end
```

This code is used in section 236.

267. The lower-case version of this database key must correspond to one in *cite_list*, or else *all_entries* must be *true*, if this entry is to be included in the reference list. Accordingly, this module sets *store_entry*, which determines whether the relevant information for this entry is stored. The alias kludge helps make the stack space not overflow on some machines.

```
  define ex_buf3 ≡ ex_buf { an alias, used only in this module }
⟨ Check for a database key of interest 267 ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln('␣is␣a␣database␣key');
  ecart
  tmp_ptr ← buf_ptr1;
  while (tmp_ptr < buf_ptr2) do
    begin ex_buf3[tmp_ptr] ← buffer[tmp_ptr]; incr(tmp_ptr);
    end;
  lower_case(ex_buf3, buf_ptr1, token_len); { convert to 'canonical' form }
  if (all_entries) then lc_cite_loc ← str_lookup(ex_buf3, buf_ptr1, token_len, lc_cite_ilk, do_insert)
  else lc_cite_loc ← str_lookup(ex_buf3, buf_ptr1, token_len, lc_cite_ilk, dont_insert);
  if (hash_found) then
    begin entry_cite_ptr ← ilk_info[ilk_info[lc_cite_loc]];
    ⟨ Check for a duplicate or crossref-matching database key 268 ⟩;
    end;
  store_entry ← true; { unless (¬hash_found) ∧ (¬all_entries) }
  if (all_entries) then ⟨ Put this cite key in its place 272 ⟩
  else if (¬hash_found) then store_entry ← false; { no such cite key exists on cite_list }
  if (store_entry) then ⟨ Make sure this entry is ok before proceeding 273 ⟩;
  end
```

This code is used in section 266.

268. It's illegal to have two (or more) entries with the same database key (even if there are case differrences), and we skip the rest of the entry for such a repeat occurrence. Also, we make this entry's database key the official *cite_list* key if it's on *cite_list* only because of cross references.

```
⟨ Check for a duplicate or crossref-matching database key 268 ⟩ ≡
  begin if ((¬all_entries) ∨ (entry_cite_ptr < all_marker) ∨ (entry_cite_ptr ≥ old_num_cites)) then
    begin if (type_list[entry_cite_ptr] = empty) then
      begin ⟨ Make sure this entry's database key is on cite_list 269 ⟩;
      goto first_time_entry;
      end;
    end
  else if (¬entry_exists[entry_cite_ptr]) then
      begin ⟨ Find the lower-case equivalent of the cite_info key 270 ⟩;
      if (lc_xcite_loc = lc_cite_loc) then goto first_time_entry;
      end;
        { oops—repeated entry—issue a reprimand }
  if (type_list[entry_cite_ptr] = empty) then confusion('The␣cite␣list␣is␣messed␣up');
  bib_err('Repeated␣entry');
first_time_entry:    { note that when we leave normally, hash_found is true }
  end
```

This code is used in section 267.

269. An entry that's on *cite_list* only because of cross referencing must have its database key (rather than one of the `crossref` keys) as the official *cite_list* string. Here's where we assure that. The variable *hash_found* is *true* upon entrance to and exit from this module.

```
⟨ Make sure this entry's database key is on cite_list 269 ⟩ ≡
  begin if ((¬all_entries) ∧ (entry_cite_ptr ≥ old_num_cites)) then
    begin cite_loc ← str_lookup(buffer, buf_ptr1, token_len, cite_ilk, do_insert);
    if (¬hash_found) then
      begin    { it's not on cite_list—put it there }
      ilk_info[lc_cite_loc] ← cite_loc; ilk_info[cite_loc] ← entry_cite_ptr;
      cite_list[entry_cite_ptr] ← hash_text[cite_loc];
      hash_found ← true;    { restore this value for later use }
      end;
    end;
  end
```

This code is used in section 268.

270. This module, a simpler version of the *find_cite_locs_for_this_cite_key* function, exists primarily to compute *lc_xcite_loc*. When this code is executed we have $(all\_entries) \wedge (entry\_cite\_ptr \geq all\_marker) \wedge (\neg entry\_exists[entry\_cite\_ptr])$. The alias kludge helps make the stack space not overflow on some machines.

```
define ex_buf4 ≡ ex_buf   { aliases, used only }
define ex_buf4_ptr ≡ ex_buf_ptr   { in this module }
```

⟨ Find the lower-case equivalent of the *cite_info* key 270 ⟩ ≡

```
begin ex_buf4_ptr ← 0; tmp_ptr ← str_start[cite_info[entry_cite_ptr]];
tmp_end_ptr ← str_start[cite_info[entry_cite_ptr] + 1];
while (tmp_ptr < tmp_end_ptr) do
  begin ex_buf4[ex_buf4_ptr] ← str_pool[tmp_ptr]; incr(ex_buf4_ptr); incr(tmp_ptr);
  end;
lower_case(ex_buf4, 0, length(cite_info[entry_cite_ptr]));   { convert to 'canonical' form }
lc_xcite_loc ← str_lookup(ex_buf4, 0, length(cite_info[entry_cite_ptr]), lc_cite_ilk, dont_insert);
if (¬hash_found) then cite_key_disappeared_confusion;
end
```

This code is used in section 268.

271. Here's another bug complaint.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure cite_key_disappeared_confusion;
  begin confusion('A␣cite␣key␣disappeared');
  end;
```

272. This module, which gets executed only when *all_entries* is *true*, does one of three things, depending on whether or not, and where, the cite key appears on *cite_list*: If it's on *cite_list* before *all_marker*, there's nothing to be done; if it's after *all_marker*, it must be reinserted (at the current place) and we must note that its corresponding entry exists; and if it's not on *cite_list* at all, it must be inserted for the first time. The **goto** construct must stay as is, partly because some PASCAL compilers might complain if "∧" were to connect the two boolean expressions (since *entry_cite_ptr* could be uninitialized when *hash_found* is *false*).

⟨ Put this cite key in its place 272 ⟩ ≡

```
begin if (hash_found) then
  begin if (entry_cite_ptr < all_marker) then goto cite_already_set   { that is, do nothing }
  else begin entry_exists[entry_cite_ptr] ← true; cite_loc ← ilk_info[lc_cite_loc];
    end;
  end
else begin   { this is a new key }
  cite_loc ← str_lookup(buffer, buf_ptr1, token_len, cite_ilk, do_insert);
  if (hash_found) then hash_cite_confusion;
  end;
entry_cite_ptr ← cite_ptr; add_database_cite(cite_ptr);   { this increments cite_ptr }
cite_already_set: end
```

This code is used in section 267.

273. We must give a warning if this entry type doesn't exist. Also, we point the appropriate entry of *type_list* to the entry type just read above.

For SCRIBE compatibility, the code to give a warning for a case mismatch between a cite key and a database key has been commented out. In fact, SCRIBE is the reason that it doesn't produce an error message outright. (Note: Case mismatches between two cite keys produce full-blown errors.)

```
⟨ Make sure this entry is ok before proceeding 273 ⟩ ≡
  begin @{dummy_loc ← str_lookup(buffer, buf_ptr1, token_len, cite_ilk, dont_insert);
  if (¬hash_found) then { give a warning if there is a case difference }
    begin print('Warning--case␣mismatch,␣database␣key␣"'); print_token; print('",␣cite␣key␣"');
    print_pool_str(cite_list[entry_cite_ptr]); bib_warn_newline('"');
    end;
  @}
  if (type_exists) then type_list[entry_cite_ptr] ← entry_type_loc
  else begin type_list[entry_cite_ptr] ← undefined; print('Warning--entry␣type␣for␣"'); print_token;
    bib_warn_newline('"␣isn''t␣style-file␣defined');
    end;
  end
```

This code is used in section 267.

274. This module reads a *comma* and a field as many times as it can, and then reads a *right_outer_delim*, ending the current entry.

```
⟨ Scan the entry's list of fields 274 ⟩ ≡
  begin while (scan_char ≠ right_outer_delim) do
    begin if (scan_char ≠ comma) then bib_one_of_two_expected_err(comma, right_outer_delim);
    incr(buf_ptr2); { skip over the comma }
    eat_bib_white_and_eof_check;
    if (scan_char = right_outer_delim) then goto loop_exit;
    ⟨ Get the next field name 275 ⟩;
    eat_bib_white_and_eof_check;
    if (¬scan_and_store_the_field_value_and_eat_white) then return;
    end;
loop_exit: incr(buf_ptr2); { skip over the right_outer_delim }
  end
```

This code is used in section 236.

275. This module reads a field name; its contents won't be stored unless it was declared in the `.bst` file and *store_entry* = *true*.

⟨ Get the next field name 275 ⟩ ≡
```
begin scan_identifier(equals_sign, equals_sign, equals_sign); bib_identifier_scan_check('a␣field␣name');
trace trace_pr_token; trace_pr_ln('␣is␣a␣field␣name');
ecart
store_field ← false;
if (store_entry) then
  begin lower_case(buffer, buf_ptr1, token_len);  { ignore case differences }
  field_name_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, dont_insert);
  if (hash_found) then
    if (fn_type[field_name_loc] = field) then
      store_field ← true;  { field name was pre-defined or .bst-declared }
  end;
eat_bib_white_and_eof_check;
if (scan_char ≠ equals_sign) then bib_equals_sign_expected_err;
incr(buf_ptr2);  { skip over the equals_sign }
end
```
This code is used in section 274.

276. This gets things ready for further `.bst` processing.

⟨ Final initialization for processing the entries 276 ⟩ ≡
```
begin num_cites ← cite_ptr;  { to include database and crossref cite keys, too }
num_preamble_strings ← preamble_ptr;  { number of preamble commands seen }
⟨ Add cross-reference information 277 ⟩;
⟨ Subtract cross-reference information 279 ⟩;
⟨ Remove missing entries or those cross referenced too few times 283 ⟩;
⟨ Initialize the int_entry_vars 287 ⟩;
⟨ Initialize the str_entry_vars 288 ⟩;
⟨ Initialize the sorted_cites 289 ⟩;
end
```
This code is used in section 223.

277. Now we update any entry (here called a *child* entry) that cross referenced another (here called a *parent* entry); this cross referencing occurs when the child's `crossref` field (value) consists of the parent's database key. To do the update, we replace the child's *missing* fields by the corresponding fields of the parent. Also, we make sure the `crossref` field contains the case-correct version. Finally, although it is technically illegal to nest cross references, and although we give a warning (a few modules hence) when someone tries, we do what we can to accommodate the attempt.

```
⟨ Add cross-reference information 277 ⟩ ≡
  begin cite_ptr ← 0;
  while (cite_ptr < num_cites) do
    begin field_ptr ← cite_ptr * num_fields + crossref_num;
    if (field_info[field_ptr] ≠ missing) then
      if (find_cite_locs_for_this_cite_key(field_info[field_ptr])) then
        begin cite_loc ← ilk_info[lc_cite_loc]; field_info[field_ptr] ← hash_text[cite_loc];
        cite_parent_ptr ← ilk_info[cite_loc]; field_ptr ← cite_ptr * num_fields + num_pre_defined_fields;
        field_end_ptr ← field_ptr − num_pre_defined_fields + num_fields;
        field_parent_ptr ← cite_parent_ptr * num_fields + num_pre_defined_fields;
        while (field_ptr < field_end_ptr) do
          begin if (field_info[field_ptr] = missing) then field_info[field_ptr] ← field_info[field_parent_ptr];
          incr(field_ptr); incr(field_parent_ptr);
          end;
        end;
    incr(cite_ptr);
    end;
  end
```

This code is used in section 276.

278. Occasionally we need to figure out the hash-table location of a given cite-key string and its lower-case equivalent. This function does that. To perform the task it needs to borrow a buffer, a need that gives rise to the alias kludge—it helps make the stack space not overflow on some machines (and while it's at it, it'll borrow a pointer, too). Finally, the function returns *true* if the cite key exists on *cite_list*, and its sets *cite_hash_found* according to whether or not it found the actual version (before *lower_case*ing) of the cite key; however, its *raison d'être* (literally, "to eat a raisin") is to compute *cite_loc* and *lc_cite_loc*.

```
  define ex_buf5 ≡ ex_buf   { aliases, used only }
  define ex_buf5_ptr ≡ ex_buf_ptr   { in this module }

⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
function find_cite_locs_for_this_cite_key(cite_str : str_number): boolean;
  begin ex_buf5_ptr ← 0; tmp_ptr ← str_start[cite_str]; tmp_end_ptr ← str_start[cite_str + 1];
  while (tmp_ptr < tmp_end_ptr) do
    begin ex_buf5[ex_buf5_ptr] ← str_pool[tmp_ptr]; incr(ex_buf5_ptr); incr(tmp_ptr);
    end;
  cite_loc ← str_lookup(ex_buf5, 0, length(cite_str), cite_ilk, dont_insert); cite_hash_found ← hash_found;
  lower_case(ex_buf5, 0, length(cite_str));   { convert to 'canonical' form }
  lc_cite_loc ← str_lookup(ex_buf5, 0, length(cite_str), lc_cite_ilk, dont_insert);
  if (hash_found) then find_cite_locs_for_this_cite_key ← true
  else find_cite_locs_for_this_cite_key ← false;
  end;
```

279. Here we remove the `crossref` field value for each child whose parent was cross referenced too few times. We also issue any necessary warnings arising from a bad cross reference.

```
⟨ Subtract cross-reference information 279 ⟩ ≡
  begin cite_ptr ← 0;
  while (cite_ptr < num_cites) do
    begin field_ptr ← cite_ptr * num_fields + crossref_num;
    if (field_info[field_ptr] ≠ missing) then
      if (¬find_cite_locs_for_this_cite_key(field_info[field_ptr])) then
        begin    { the parent is not on cite_list }
        if (cite_hash_found) then hash_cite_confusion;
        nonexistent_cross_reference_error; field_info[field_ptr] ← missing;    { remove the crossref ptr }
        end
      else begin    { the parent exists on cite_list }
        if (cite_loc ≠ ilk_info[lc_cite_loc]) then hash_cite_confusion;
        cite_parent_ptr ← ilk_info[cite_loc];
        if (type_list[cite_parent_ptr] = empty) then
          begin nonexistent_cross_reference_error;
          field_info[field_ptr] ← missing;    { remove the crossref ptr }
          end
        else begin    { the parent exists in the database too }
          field_parent_ptr ← cite_parent_ptr * num_fields + crossref_num;
          if (field_info[field_parent_ptr] ≠ missing) then ⟨ Complain about a nested cross reference 282 ⟩;
          if ((¬all_entries) ∧ (cite_parent_ptr ≥ old_num_cites) ∧ (cite_info[cite_parent_ptr] < min_crossrefs))
              then
            field_info[field_ptr] ← missing;    { remove the crossref ptr }
          end;
        end;
    incr(cite_ptr);
    end;
  end
```

This code is used in section 276.

280. This procedure exists to save space, since it's used twice—once for each of the two succeeding modules.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure bad_cross_reference_print(s : str_number);
  begin print('--entry␣"'); print_pool_str(cur_cite_str); print_ln('"'); print('refers␣to␣entry␣"');
  print_pool_str(s);
  end;
```

281. When an entry being cross referenced doesn't exist on *cite_list*, we complain.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure nonexistent_cross_reference_error;
  begin print('A␣bad␣cross␣reference-'); bad_cross_reference_print(field_info[field_ptr]);
  print_ln('",␣which␣doesn''t␣exist'); mark_error;
  end;
```

282. We also complain when an entry being cross referenced has a non*missing* `crossref` field itself, but this one is just a warning, not a full-blown error.

⟨ Complain about a nested cross reference 282 ⟩ ≡
```
begin print(´Warning--you´´ve␣nested␣cross␣references´);
bad_cross_reference_print(cite_list[cite_parent_ptr]); print_ln(´",␣which␣also␣refers␣to␣something´);
mark_warning;
end
```
This code is used in section 279.

283. We remove (and give a warning for) each cite key on the original *cite_list* without a corresponding database entry. And we remove any entry that was included on *cite_list* only because it was cross referenced, yet was cross referenced fewer than *min_crossrefs* times. Throughout this module, *cite_ptr* points to the next cite key to be checked and *cite_xptr* points to the next permanent spot on *cite_list*.

⟨ Remove missing entries or those cross referenced too few times 283 ⟩ ≡
```
begin cite_ptr ← 0;
while (cite_ptr < num_cites) do
  begin if (type_list[cite_ptr] = empty) then print_missing_entry(cur_cite_str)
  else if ((all_entries) ∨ (cite_ptr < old_num_cites) ∨ (cite_info[cite_ptr] ≥ min_crossrefs)) then
      begin if (cite_ptr > cite_xptr) then ⟨ Slide this cite key down to its permanent spot 285 ⟩;
      incr(cite_xptr);
      end;
  incr(cite_ptr);
  end;
num_cites ← cite_xptr;
if (all_entries) then ⟨ Complain about missing entries whose cite keys got overwritten 286 ⟩;
end
```
This code is used in section 276.

284. When a cite key on the original *cite_list* (or added to *cite_list* because of cross referencing) didn't appear in the database, complain.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
```
procedure print_missing_entry(s : str_number);
  begin print(´Warning--I␣didn´´t␣find␣a␣database␣entry␣for␣"´); print_pool_str(s); print_ln(´"´);
  mark_warning;
  end;
```

285. We have to move to its final resting place all the entry information associated with the exact location in *cite_list* of this cite key.

⟨ Slide this cite key down to its permanent spot 285 ⟩ ≡
```
begin cite_list[cite_xptr] ← cite_list[cite_ptr]; type_list[cite_xptr] ← type_list[cite_ptr];
if (¬find_cite_locs_for_this_cite_key(cite_list[cite_ptr])) then cite_key_disappeared_confusion;
if ((¬cite_hash_found) ∨ (cite_loc ≠ ilk_info[lc_cite_loc])) then hash_cite_confusion;
ilk_info[cite_loc] ← cite_xptr;
field_ptr ← cite_xptr * num_fields; field_end_ptr ← field_ptr + num_fields; tmp_ptr ← cite_ptr * num_fields;
while (field_ptr < field_end_ptr) do
  begin field_info[field_ptr] ← field_info[tmp_ptr]; incr(field_ptr); incr(tmp_ptr);
  end;
end
```
This code is used in section 283.

286. We need this module only when we're including the whole database. It's for missing entries whose cite key originally resided in *cite_list* at a spot that another cite key (might have) claimed.

```
⟨ Complain about missing entries whose cite keys got overwritten 286 ⟩ ≡
  begin cite_ptr ← all_marker;
  while (cite_ptr < old_num_cites) do
    begin if (¬entry_exists[cite_ptr]) then print_missing_entry(cite_info[cite_ptr]);
    incr(cite_ptr);
    end;
  end
```

This code is used in section 283.

287. This module initializes all *int_entry_var*s of all entries to 0, the value to which all integers are initialized.

```
⟨ Initialize the int_entry_vars 287 ⟩ ≡
  begin if (num_ent_ints * num_cites > max_ent_ints) then
    begin print(num_ent_ints * num_cites, ´:␣´);
    overflow(´total␣number␣of␣integer␣entry-variables␣´, max_ent_ints);
    end;
  int_ent_ptr ← 0;
  while (int_ent_ptr < num_ent_ints * num_cites) do
    begin entry_ints[int_ent_ptr] ← 0; incr(int_ent_ptr);
    end;
  end
```

This code is used in section 276.

288. This module initializes all *str_entry_var*s of all entries to the null string, the value to which all strings are initialized.

```
⟨ Initialize the str_entry_vars 288 ⟩ ≡
  begin if (num_ent_strs * num_cites > max_ent_strs) then
    begin print(num_ent_strs * num_cites, ´:␣´);
    overflow(´total␣number␣of␣string␣entry-variables␣´, max_ent_strs);
    end;
  str_ent_ptr ← 0;
  while (str_ent_ptr < num_ent_strs * num_cites) do
    begin entry_strs[str_ent_ptr][0] ← end_of_string; incr(str_ent_ptr);
    end;
  end
```

This code is used in section 276.

289. The array *sorted_cites* initially specifies that the entries are to be processed in order of cite-key occurrence. The `sort` command may change this to whatever it likes (which, we hope, is whatever the style-designer instructs it to like). We make *sorted_cites* an alias to save space; this works fine because we're done with *cite_info*.

```
  define sorted_cites ≡ cite_info   { an alias used for the rest of the program }
⟨ Initialize the sorted_cites 289 ⟩ ≡
  begin cite_ptr ← 0;
  while (cite_ptr < num_cites) do
    begin sorted_cites[cite_ptr] ← cite_ptr; incr(cite_ptr);
    end;
  end
```

This code is used in section 276.

290. Executing the style file. This part of the program produces the output by executing the `.bst`-file commands `execute`, `iterate`, `reverse`, and `sort`. To do this it uses a stack (consisting of the two arrays *lit_stack* and *lit_stk_type*) for storing literals, a buffer *ex_buf* for manipulating strings, and an array *sorted_cites* for holding pointers to the sorted cite keys (*sorted_cites* is an alias of *cite_info*).

⟨ Globals in the outer block 16 ⟩ +≡
lit_stack: **array** [*lit_stk_loc*] **of** *integer*; { the literal function stack }
lit_stk_type: **array** [*lit_stk_loc*] **of** *stk_type*; { their corresponding types }
lit_stk_ptr: *lit_stk_loc*; { points just above the top of the stack }
cmd_str_ptr: *str_number*; { stores value of *str_ptr* during execution }
ent_chr_ptr: 0 .. *ent_str_size*; { points at a *str_entry_var* character }
glob_chr_ptr: 0 .. *glob_str_size*; { points at a *str_global_var* character }
ex_buf: *buf_type*; { a buffer for manipulating strings }
ex_buf_ptr: *buf_pointer*; { general *ex_buf* location }
ex_buf_length: *buf_pointer*; { the length of the current string in *ex_buf* }
out_buf: *buf_type*; { the `.bbl` output buffer }
out_buf_ptr: *buf_pointer*; { general *out_buf* location }
out_buf_length: *buf_pointer*; { the length of the current string in *out_buf* }
mess_with_entries: *boolean*; { *true* if functions can use entry info }
sort_cite_ptr: *cite_number*; { a loop index for the sorted cite keys }
sort_key_num: *str_ent_loc*; { index for the *str_entry_var* `sort.key$` }
brace_level: *integer*; { the brace nesting depth within a string }

291. Where *lit_stk_loc* is a stack location, and where *stk_type* gives one of the three types of literals (an integer, a string, or a function) or a special marker. If a *lit_stk_type* element is a *stk_int* then the corresponding *lit_stack* element is an integer; if a *stk_str*, then a pointer to a *str_pool* string; and if a *stk_fn*, then a pointer to the function's hash-table location. However, if the literal should have been a *stk_str* that was the value of a field that happened to be *missing*, then the special value *stk_field_missing* goes on the stack instead; its corresponding *lit_stack* element is a pointer to the field-name's string. Finally, *stk_empty* is the type of a literal popped from an empty stack.

define *stk_int* = 0 { an integer literal }
define *stk_str* = 1 { a string literal }
define *stk_fn* = 2 { a function literal }
define *stk_field_missing* = 3 { a special marker: a field value was missing }
define *stk_empty* = 4 { another: the stack was empty when this was popped }
define *last_lit_type* = 4 { the same number as on the line above }

⟨ Types in the outer block 22 ⟩ +≡
lit_stk_loc = 0 .. *lit_stk_size*; { the stack range }
stk_type = 0 .. *last_lit_type*; { the literal types }

292. And the first output line requires this initialization.

⟨ Set initial values of key variables 20 ⟩ +≡
out_buf_length ← 0;

293. When there's an error while executing `.bst` functions, what we do depends on whether the function is messing with the entries. Furthermore this error is serious enough to classify as an *error_message* instead of a *warning_message*. These messages (that is, from *bst_ex_warn*) are meant both for the user and for the style designer while debugging.

```
define bst_ex_warn(#) ≡
        begin  { error while executing some function }
        print(#); bst_ex_warn_print;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bst_ex_warn_print;
  begin if (mess_with_entries) then
    begin print(´␣for␣entry␣´); print_pool_str(cur_cite_str);
    end;
  print_newline; print(´while␣executing-´); bst_ln_num_print; mark_error;
  end;
```

294. When an error is so harmless, we print a *warning_message* instead of an *error_message*.

```
define bst_mild_ex_warn(#) ≡
        begin  { error while executing some function }
        print(#); bst_mild_ex_warn_print;
        end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bst_mild_ex_warn_print;
  begin if (mess_with_entries) then
    begin print(´␣for␣entry␣´); print_pool_str(cur_cite_str);
    end;
  print_newline; bst_warn(´while␣executing´);  { This does the mark_warning }
  end;
```

295. It's illegal to mess with the entry information at certain times; here's a complaint for these times.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡

```
procedure bst_cant_mess_with_entries_print;
  begin bst_ex_warn(´You␣can´´t␣mess␣with␣entries␣here´);
  end;
```

296. This module executes a single specified function once. It can't do anything with the entries.

⟨ Perform an `execute` command 296 ⟩ ≡

```
  begin init_command_execution; mess_with_entries ← false; execute_fn(fn_loc);
  check_command_execution;
  end
```

This code is used in section 178.

297. This module iterates a single specified function for all entries specified by *cite_list*.

```
⟨ Perform an iterate command 297 ⟩ ≡
  begin init_command_execution; mess_with_entries ← true; sort_cite_ptr ← 0;
  while (sort_cite_ptr < num_cites) do
    begin cite_ptr ← sorted_cites[sort_cite_ptr];
    trace trace_pr_pool_str(hash_text[fn_loc]); trace_pr('␣to␣be␣iterated␣on␣');
    trace_pr_pool_str(cur_cite_str); trace_pr_newline;
    ecart
    execute_fn(fn_loc); check_command_execution; incr(sort_cite_ptr);
    end;
  end
```

This code is used in section 203.

298. This module iterates a single specified function for all entries specified by *cite_list*, but does it in reverse order.

```
⟨ Perform a reverse command 298 ⟩ ≡
  begin init_command_execution; mess_with_entries ← true;
  if (num_cites > 0) then
    begin sort_cite_ptr ← num_cites;
    repeat decr(sort_cite_ptr); cite_ptr ← sorted_cites[sort_cite_ptr];
      trace trace_pr_pool_str(hash_text[fn_loc]); trace_pr('␣to␣be␣iterated␣in␣reverse␣on␣');
      trace_pr_pool_str(cur_cite_str); trace_pr_newline;
      ecart
      execute_fn(fn_loc); check_command_execution;
    until (sort_cite_ptr = 0);
    end;
  end
```

This code is used in section 212.

299. This module sorts the entries based on `sort.key$`; it is a stable sort.

```
⟨ Perform a sort command 299 ⟩ ≡
  begin trace trace_pr_ln('Sorting␣the␣entries');
  ecart
  if (num_cites > 1) then quick_sort(0, num_cites − 1);
  trace trace_pr_ln('Done␣sorting');
  ecart
  end
```

This code is used in section 214.

300. These next two procedures (actually, one procedures and one function, but who's counting) are subroutines for *quick_sort*, which follows. The *swap* procedure exchanges the two elements its arguments point to.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure swap(swap1, swap2 : cite_number);
  var innocent_bystander: cite_number; { the temporary element in an exchange }
  begin innocent_bystander ← sorted_cites[swap2]; sorted_cites[swap2] ← sorted_cites[swap1];
  sorted_cites[swap1] ← innocent_bystander;
  end;
```

301. The function *less_than* compares the two `sort.key$`s indirectly pointed to by its arguments and returns *true* if the first argument's `sort.key$` is lexicographically less than the second's (that is, alphabetically earlier). In case of ties the function compares the indices *arg1* and *arg2*, which are assumed to be different, and returns *true* if the first is smaller. This function uses *ASCII_code*s to compare, so it might give "interesting" results when handling nonletters.

```
define compare_return(#) ≡
        begin   { the compare is finished }
        less_than ← #; return;
        end
```

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
function less_than(arg1, arg2 : cite_number): boolean;
  label exit;
  var char_ptr: 0 .. ent_str_size;   { character index into compared strings }
    ptr1, ptr2: str_ent_loc;   { the two sort.key$ pointers }
    char1, char2: ASCII_code;   { the two characters being compared }
  begin ptr1 ← arg1 * num_ent_strs + sort_key_num; ptr2 ← arg2 * num_ent_strs + sort_key_num;
  char_ptr ← 0;
  loop
    begin char1 ← entry_strs[ptr1][char_ptr]; char2 ← entry_strs[ptr2][char_ptr];
    if (char1 = end_of_string) then
      if (char2 = end_of_string) then
        if (arg1 < arg2) then compare_return(true)
        else if (arg1 > arg2) then compare_return(false)
          else   { arg1 = arg2 }
    confusion('Duplicate␣sort␣key')
  else   { char2 ≠ end_of_string }
    compare_return(true)
  else   { char1 ≠ end_of_string }
    if (char2 = end_of_string) then compare_return(false)
    else if (char1 < char2) then compare_return(true)
      else if (char1 > char2) then compare_return(false);
    incr(char_ptr);
    end;
exit: end;
```

302. The recursive procedure *quick_sort* sorts the entries indirectly pointed to by the *sorted_cites* elements between *left_end* and *right_end*, inclusive, based on the value of the *str_entry_var* `sort.key$`. It's a fairly standard quicksort (for example, see Algorithm 5.2.2Q in *The Art of Computer Programming*), but uses the median-of-three method to choose the partition element just in case the entries are already sorted (or nearly sorted—humans and ASCII might have different ideas on lexicographic ordering); it is a stable sort. This code generally prefers clarity to assembler-type execution-time efficiency since *cite_list*s will rarely be huge.

The value *short_list*, which must be at least $2 * end\_offset + 2$ for this code to work, tells us the list-length at which the list is small enough to warrant switching over to straight insertion sort from the recursive quicksort. The values here come from modest empirical tests aimed at minimizing, for large *cite_list*s (five hundred or so), the number of comparisons (between keys) plus the number of calls to *quick_sort*. The value *end_offset* must be positive; this helps avoid n^2 behavior observed when the list starts out nearly, but not completely, sorted (and fairly frequently large *cite_list*s come from entire databases, which fairly frequently are nearly sorted).

```
define short_list = 10  { use straight insertion sort at or below this length }
define end_offset = 4  { the index end-offsets for choosing a median-of-three }
⟨ Check the "constant" values for consistency 17 ⟩ +≡
  if (short_list < 2 * end_offset + 2) then bad ← 100 * bad + 22;
```

303. Here's the actual procedure.

```
define next_insert = 24  { now insert the next element }
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure quick_sort(left_end, right_end : cite_number);
  label next_insert;
  var left, right: cite_number;  { two general sorted_cites pointers }
    insert_ptr: cite_number;  { the to-be-(straight)-inserted element }
    middle: cite_number;  { the (left_end + right_end) div 2 element }
    partition: cite_number;  { the median-of-three partition element }
  begin trace trace_pr_ln('Sorting␣', left_end : 0, '␣through␣', right_end : 0);
  ecart
  if (right_end − left_end < short_list) then ⟨ Do a straight insertion sort 304 ⟩
  else begin ⟨ Draw out the median-of-three partition element 305 ⟩;
    ⟨ Do the partitioning and the recursive calls 306 ⟩;
    end;
  end;
```

304. This code sorts the entries between *left_end* and *right_end* when the difference is less than *short_list*. Each iteration of the outer loop inserts the element indicated by *insert_ptr* into its proper place among the (sorted) elements from *left_end* through *insert_ptr* − 1.

```
⟨ Do a straight insertion sort 304 ⟩ ≡
  begin for insert_ptr ← left_end + 1 to right_end do
    begin for right ← insert_ptr downto left_end + 1 do
      begin if (less_than(sorted_cites[right − 1], sorted_cites[right])) then goto next_insert;
      swap(right − 1, right);
      end;
  next_insert: end;
  end
```

This code is used in section 303.

305. Now we find the median of the three `sort.key$`s to which the three elements *sorted_cites*[*left_end* + *end_offset*], *sorted_cites*[*right_end*] − *end_offset*, and *sorted_cites*[(*left_end* + *right_end*) **div** 2] point (a nonzero *end_offset* avoids using as the leftmost of the three elements the one that was swapped there when the old partition element was swapped into its final spot; this turns out to avoid n^2 behavior when the list is nearly sorted to start with). This code determines which of the six possible permutations we're dealing with and moves the median element to *left_end*. The comments next to the *swap* actions give the known orderings of the corresponding elements of *sorted_cites* before the action.

```
⟨ Draw out the median-of-three partition element 305 ⟩ ≡
  begin left ← left_end + end_offset; middle ← (left_end + right_end) div 2; right ← right_end − end_offset;
  if (less_than(sorted_cites[left], sorted_cites[middle])) then
    if (less_than(sorted_cites[middle], sorted_cites[right])) then   { left < middle < right }
      swap(left_end, middle)
    else if (less_than(sorted_cites[left], sorted_cites[right])) then   { left < right < middle }
        swap(left_end, right)
      else   { right < left < middle }
  swap(left_end, left)
else if (less_than(sorted_cites[right], sorted_cites[middle])) then   { right < middle < left }
    swap(left_end, middle)
  else if (less_than(sorted_cites[right], sorted_cites[left])) then   { middle < right < left }
      swap(left_end, right)
    else   { middle < left < right }
  swap(left_end, left);
  end
```

This code is used in section 303.

306. This module uses the median-of-three computed above to partition the elements into those less than and those greater than the median. Equal `sort.key$`s are sorted by order of occurrence (in *cite_list*).

```
⟨ Do the partitioning and the recursive calls 306 ⟩ ≡
  begin partition ← sorted_cites[left_end]; left ← left_end + 1; right ← right_end;
  repeat while (less_than(sorted_cites[left], partition)) do incr(left);
    while (less_than(partition, sorted_cites[right])) do decr(right);
          { now sorted_cites[right] < partition < sorted_cites[left] }
    if (left < right) then
      begin swap(left, right); incr(left); decr(right);
      end;
  until (left = right + 1);   { pointers have crossed }
  swap(left_end, right);   { restoring the partition element to its rightful place }
  quick_sort(left_end, right − 1); quick_sort(left, right_end);
  end
```

This code is used in section 303.

307. Ok, that's it for sorting; now we'll play with the literal stack. This procedure pushes a literal onto the stack, checking for stack overflow.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ ≡
procedure push_lit_stk(push_lt : integer; push_type : stk_type);
    trace
  var dum_ptr: lit_stk_loc;  { used just as an index variable }
    ecart
    begin lit_stack[lit_stk_ptr] ← push_lt; lit_stk_type[lit_stk_ptr] ← push_type;
    trace for dum_ptr ← 0 to lit_stk_ptr do trace_pr(´␣␣´);
    trace_pr(´Pushing␣´);
    case (lit_stk_type[lit_stk_ptr]) of
    stk_int: trace_pr_ln(lit_stack[lit_stk_ptr] : 0);
    stk_str: begin trace_pr(´"´); trace_pr_pool_str(lit_stack[lit_stk_ptr]); trace_pr_ln(´"´);
      end;
    stk_fn: begin trace_pr(´`´); trace_pr_pool_str(hash_text[lit_stack[lit_stk_ptr]]); trace_pr_ln(´´´);
      end;
    stk_field_missing: begin trace_pr(´missing␣field␣`´); trace_pr_pool_str(lit_stack[lit_stk_ptr]);
      trace_pr_ln(´´´);
      end;
    stk_empty: trace_pr_ln(´a␣bad␣literal--popped␣from␣an␣empty␣stack´);
    othercases unknwn_literal_confusion
    endcases;
    ecart
    if (lit_stk_ptr = lit_stk_size) then overflow(´literal-stack␣size␣´, lit_stk_size);
    incr(lit_stk_ptr);
    end;
```

See also sections 309, 312, 314, 315, 316, 317, 318, 320, 322·10^T $s342$.

This code is used in section 12.

308. This macro pushes the last thing, necessarily a string, that was popped. And this module, along with others that push the literal stack without explicitly calling *push_lit_stack*, have an index entry under "push the literal stack"; these implicit pushes collectively speed up the program by about ten percent.

```
define repush_string ≡
        begin if (lit_stack[lit_stk_ptr] ≥ cmd_str_ptr) then unflush_string;
        incr(lit_stk_ptr);
        end
```

309. This procedure pops the stack, checking for, and trying to recover from, stack underflow. (Actually, this procedure is really a function, since it returns the two values through its **var** parameters.) Also, if the literal being popped is a *stk_str* that's been created during the execution of the current `.bst` command, pop it from *str_pool* as well (it will be the string corresponding to *str_ptr* − 1). Note that when this happens, the string is no longer 'officially' available so that it must be used before anything else is added to *str_pool*.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure pop_lit_stk(var pop_lit : integer; var pop_type : stk_type);
  begin if (lit_stk_ptr = 0) then
    begin bst_ex_warn('You␣can''t␣pop␣an␣empty␣literal␣stack');
    pop_type ← stk_empty;  { this is an error recovery attempt }
    end
  else begin decr(lit_stk_ptr); pop_lit ← lit_stack[lit_stk_ptr]; pop_type ← lit_stk_type[lit_stk_ptr];
    if (pop_type = stk_str) then
      if (pop_lit ≥ cmd_str_ptr) then
        begin if (pop_lit ≠ str_ptr − 1) then confusion('Nontop␣top␣of␣string␣stack');
        flush_string;
        end;
    end;
  end;
```

310. More bug complaints, this time about bad literals.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure illegl_literal_confusion;
  begin confusion('Illegal␣literal␣type');
  end;
procedure unknwn_literal_confusion;
  begin confusion('Unknown␣literal␣type');
  end;
```

311. Occasionally we'll want to know what's on the literal stack. Here we print out a stack literal, giving its type. This procedure should never be called after popping an empty stack.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure print_stk_lit(stk_lt : integer; stk_tp : stk_type);
  begin case (stk_tp) of
  stk_int: print(stk_lt : 0, '␣is␣an␣integer␣literal');
  stk_str: begin print('"'); print_pool_str(stk_lt); print('"␣is␣a␣string␣literal');
    end;
  stk_fn: begin print('`'); print_pool_str(hash_text[stk_lt]); print('''␣is␣a␣function␣literal');
    end;
  stk_field_missing: begin print('`'); print_pool_str(stk_lt); print('''␣is␣a␣missing␣field');
    end;
  stk_empty: illegl_literal_confusion;
  othercases unknwn_literal_confusion
  endcases;
  end;
```

312. This procedure appropriately chastises the style designer; however, if the wrong literal came from popping an empty stack, the procedure *pop_lit_stack* will have already done the chastising (because this procedure is called only after popping the stack) so there's no need for more.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure print_wrong_stk_lit(stk_lt : integer; stk_tp1, stk_tp2 : stk_type);
  begin if (stk_tp1 ≠ stk_empty) then
    begin print_stk_lit(stk_lt, stk_tp1);
    case (stk_tp2) of
    stk_int: print(´,␣not␣an␣integer,´);
    stk_str: print(´,␣not␣a␣string,´);
    stk_fn: print(´,␣not␣a␣function,´);
    stk_field_missing, stk_empty: illegl_literal_confusion;
    othercases unknwn_literal_confusion
    endcases; bst_ex_warn_print;
    end;
  end;
```

313. This is similar to *print_stk_lit*, but here we don't give the literal's type, and here we end with a new line. This procedure should never be called after popping an empty stack.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure print_lit(stk_lt : integer; stk_tp : stk_type);
  begin case (stk_tp) of
  stk_int: print_ln(stk_lt : 0);
  stk_str: begin print_pool_str(stk_lt); print_newline;
    end;
  stk_fn: begin print_pool_str(hash_text[stk_lt]); print_newline;
    end;
  stk_field_missing: begin print_pool_str(stk_lt); print_newline;
    end;
  stk_empty: illegl_literal_confusion;
  othercases unknwn_literal_confusion
  endcases;
  end;
```

314. This procedure pops and prints the top of the stack; when the stack is empty the procedure *pop_lit_stk* complains.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure pop_top_and_print;
  var stk_lt: integer; stk_tp: stk_type;
  begin pop_lit_stk(stk_lt, stk_tp);
  if (stk_tp = stk_empty) then print_ln(´Empty␣literal´)
  else print_lit(stk_lt, stk_tp);
  end;
```

315. This procedure pops and prints the whole stack.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure pop_whole_stack;
  begin while (lit_stk_ptr > 0) do pop_top_and_print;
  end;
```

316. At the beginning of a `.bst`-command execution we make the stack empty and record how much of *str_pool* has been used.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure init_command_execution;
  begin lit_stk_ptr ← 0;  { make the stack empty }
  cmd_str_ptr ← str_ptr;  { we'll check this when we finish command execution }
  end;
```

317. At the end of a `.bst` command-execution we check that the stack and *str_pool* are still in good shape.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure check_command_execution;
  begin if (lit_stk_ptr ≠ 0) then
    begin print_ln('ptr=', lit_stk_ptr : 0, ', stack=');  pop_whole_stack;
    bst_ex_warn('---the literal stack isn''t empty');
    end;
  if (cmd_str_ptr ≠ str_ptr) then
    begin trace print_ln('Pointer is ', str_ptr : 0, ' but should be ', cmd_str_ptr : 0);
    ecart
    confusion('Nonempty empty string stack');
    end;
  end;
```

318. This procedure adds to *str_pool* the string from *ex_buf*[0] through *ex_buf*[*ex_buf_length* − 1] if it will fit. It assumes the global variable *ex_buf_length* gives the length of the current string in *ex_buf*. It then pushes this string onto the literal stack.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure add_pool_buf_and_push;
  begin str_room(ex_buf_length);  { make sure this string will fit }
  ex_buf_ptr ← 0;
  while (ex_buf_ptr < ex_buf_length) do
    begin append_char(ex_buf[ex_buf_ptr]);  incr(ex_buf_ptr);
    end;
  push_lit_stk(make_string, stk_str);  { and push it onto the stack }
  end;
```

319. These macros append a character to *ex_buf*. Which is called depends on whether the character is known to fit.

```
define append_ex_buf_char(#) ≡
         begin ex_buf[ex_buf_ptr] ← #;  incr(ex_buf_ptr);
         end
define append_ex_buf_char_and_check(#) ≡
         begin if (ex_buf_ptr = buf_size) then buffer_overflow;
         append_ex_buf_char(#);
         end
```

320. This procedure adds to the execution buffer the given string in *str_pool* if it will fit. It assumes the global variable *ex_buf_length* gives the length of the current string in *ex_buf*, and thus also gives the location of the next character.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure add_buf_pool(p_str : str_number);
  begin p_ptr1 ← str_start[p_str]; p_ptr2 ← str_start[p_str + 1];
  if (ex_buf_length + (p_ptr2 − p_ptr1) > buf_size) then buffer_overflow;
  ex_buf_ptr ← ex_buf_length;
  while (p_ptr1 < p_ptr2) do
    begin { copy characters into the buffer }
    append_ex_buf_char(str_pool[p_ptr1]); incr(p_ptr1);
    end;
  ex_buf_length ← ex_buf_ptr;
  end;
```

321. This procedure actually writes onto the `.bbl` file a line of output (the characters from *out_buf*[0] to *out_buf*[*out_buf_length* − 1], after removing trailing *white_space* characters). It also updates *bbl_line_num*, the line counter. It writes a blank line if and only if *out_buf* is empty. The program uses this procedure in such a way that *out_buf* will be nonempty if there have been characters put in it since the most recent `newline$`.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure output_bbl_line;
  label loop_exit, exit;
  begin if (out_buf_length ≠ 0) then { the buffer's not empty }
    begin while (out_buf_length > 0) do { remove trailing white_space }
      if (lex_class[out_buf[out_buf_length − 1]] = white_space) then decr(out_buf_length)
      else goto loop_exit;
  loop_exit: if (out_buf_length = 0) then { ignore a line of just white_space }
      return;
    out_buf_ptr ← 0;
    while (out_buf_ptr < out_buf_length) do
      begin write(bbl_file, xchr[out_buf[out_buf_ptr]]); incr(out_buf_ptr);
      end;
    end;
  write_ln(bbl_file); incr(bbl_line_num); { update line number }
  out_buf_length ← 0; { make the next line empty }
exit: end;
```

322. This procedure adds to the output buffer the given string in *str_pool*. It assumes the global variable *out_buf_length* gives the length of the current string in *out_buf*, and thus also gives the location for the next character. If there are enough characters present in the output buffer, it writes one or more lines out to the `.bbl` file. It breaks a line only at a *white_space* character, and when it does, it adds two *space*s to the next output line.

```
⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
procedure add_out_pool(p_str : str_number);
  label loop1_exit, loop2_exit;
  var break_ptr: buf_pointer;  { the first character following the line break }
    end_ptr: buf_pointer;  { temporary end-of-buffer pointer }
    break_pt_found: boolean;  { a suitable white_space character }
    unbreakable_tail: boolean;  { as it contains no white_space character }
  begin p_ptr1 ← str_start[p_str]; p_ptr2 ← str_start[p_str + 1];
  if (out_buf_length + (p_ptr2 − p_ptr1) > buf_size) then overflow('output␣buffer␣size␣', buf_size);
  out_buf_ptr ← out_buf_length;
  while (p_ptr1 < p_ptr2) do
    begin  { copy characters into the buffer }
    out_buf[out_buf_ptr] ← str_pool[p_ptr1]; incr(p_ptr1); incr(out_buf_ptr);
    end;
  out_buf_length ← out_buf_ptr; unbreakable_tail ← false;
  while ((out_buf_length > max_print_line) ∧ (¬unbreakable_tail)) do ⟨ Break that line 323 ⟩;
  end;
```

323. Here we break the line by looking for a *white_space* character, backwards from *out_buf*[*max_print_line*] until *out_buf*[*min_print_line*]; we break at the *white_space* and indent the next line two *space*s. The next module handles things when there's no *white_space* character to break at. (It seems that the annoyances to the average user of a warning message when there's an output line longer than *max_print_line* outweigh the benefits, so we don't issue such warnings in the current code.)

```
⟨ Break that line 323 ⟩ ≡
  begin end_ptr ← out_buf_length; out_buf_ptr ← max_print_line; break_pt_found ← false;
  while ((lex_class[out_buf[out_buf_ptr]] ≠ white_space) ∧ (out_buf_ptr ≥ min_print_line)) do
    decr(out_buf_ptr);
  if (out_buf_ptr = min_print_line − 1) then  { no white_space character }
    ⟨ Break that unbreakably long line 324 ⟩  { (if white_space follows) }
  else break_pt_found ← true;  { hit a white_space character }
  if (break_pt_found) then
    begin out_buf_length ← out_buf_ptr; break_ptr ← out_buf_length + 1; output_bbl_line;
        { output what we can }
    out_buf[0] ← space; out_buf[1] ← space;  { start the next line with two spaces }
    out_buf_ptr ← 2; tmp_ptr ← break_ptr;
    while (tmp_ptr < end_ptr) do  { and slide the rest down }
      begin out_buf[out_buf_ptr] ← out_buf[tmp_ptr]; incr(out_buf_ptr); incr(tmp_ptr);
      end;
    out_buf_length ← end_ptr − break_ptr + 2;
    end;
  end
```

This code is used in section 322.

324. If there's no *white_space* character up through *out_buf*[*max_print_line*], we instead break the line at the first following *white_space* character, if one exists. And if, starting with that *white_space* character, there are multiple consecutive *white_space* characters, *out_buf_ptr* points to the last of them. If no *white_space* character exists, we haven't found a viable break point, so we don't break the line (yet).

```
⟨ Break that unbreakably long line 324 ⟩ ≡
  begin out_buf_ptr ← max_print_line + 1;  { break_pt_found is still false }
  while (out_buf_ptr < end_ptr) do
    if (lex_class[out_buf[out_buf_ptr]] ≠ white_space) then incr(out_buf_ptr)
    else goto loop1_exit;
loop1_exit: if (out_buf_ptr = end_ptr) then unbreakable_tail ← true  { because no white_space character }
  else  { at white_space, and out_buf_ptr < end_ptr }
  begin break_pt_found ← true;
  while (out_buf_ptr + 1 < end_ptr) do  { look for more white_space }
    if (lex_class[out_buf[out_buf_ptr + 1]] = white_space) then incr(out_buf_ptr)
          { which then points to white_space }
    else goto loop2_exit;
loop2_exit: end;
  end
```

This code is used in section 323.

325. This procedure executes a single specified function; it is the single execution-primitive that does everything (except windows, and it takes Tuesdays off).

```
⟨ execute_fn itself 325 ⟩ ≡
procedure execute_fn(ex_fn_loc : hash_loc);
  ⟨ Declarations for executing built_in functions 343 ⟩ wiz_ptr: wiz_fn_loc;  { general wiz_functions location }
  begin trace trace_pr('execute_fn␣'); trace_pr_pool_str(hash_text[ex_fn_loc]); trace_pr_ln('''');
  ecart
  case (fn_type[ex_fn_loc]) of
  built_in: ⟨ Execute a built_in function 341 ⟩;
  wiz_defined: ⟨ Execute a wiz_defined function 326 ⟩;
  int_literal: push_lit_stk(fn_info[ex_fn_loc], stk_int);
  str_literal: push_lit_stk(hash_text[ex_fn_loc], stk_str);
  field: ⟨ Execute a field 327 ⟩;
  int_entry_var: ⟨ Execute an int_entry_var 328 ⟩;
  str_entry_var: ⟨ Execute a str_entry_var 329 ⟩;
  int_global_var: push_lit_stk(fn_info[ex_fn_loc], stk_int);
  str_global_var: ⟨ Execute a str_global_var 330 ⟩;
  othercases unknwn_function_class_confusion
  endcases;
  end;
```

This code is used in section 342.

326. To execute a *wiz_defined* function, we just execute all those functions in its definition, except that the special marker *quote_next_fn* means we push the next function onto the stack.

```
⟨ Execute a wiz_defined function 326 ⟩ ≡
  begin wiz_ptr ← fn_info[ex_fn_loc];
  while (wiz_functions[wiz_ptr] ≠ end_of_def) do
    begin if (wiz_functions[wiz_ptr] ≠ quote_next_fn) then execute_fn(wiz_functions[wiz_ptr])
    else begin incr(wiz_ptr); push_lit_stk(wiz_functions[wiz_ptr], stk_fn);
      end;
    incr(wiz_ptr);
    end;
  end
```

This code is used in section 325.

327. This module pushes the string given by the field onto the literal stack unless it's *missing*, in which case it pushes a special value onto the stack.

```
⟨ Execute a field 327 ⟩ ≡
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else begin field_ptr ← cite_ptr * num_fields + fn_info[ex_fn_loc];
    if (field_info[field_ptr] = missing) then push_lit_stk(hash_text[ex_fn_loc], stk_field_missing)
    else push_lit_stk(field_info[field_ptr], stk_str);
    end
  end
```

This code is used in section 325.

328. This module pushes the integer given by an *int_entry_var* onto the literal stack.

```
⟨ Execute an int_entry_var 328 ⟩ ≡
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else push_lit_stk(entry_ints[cite_ptr * num_ent_ints + fn_info[ex_fn_loc]], stk_int);
  end
```

This code is used in section 325.

329. This module adds the string given by a *str_entry_var* to *str_pool* via the execution buffer and pushes it onto the literal stack.

```
⟨ Execute a str_entry_var 329 ⟩ ≡
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else begin str_ent_ptr ← cite_ptr * num_ent_strs + fn_info[ex_fn_loc];
    ex_buf_ptr ← 0;  { also serves as ent_chr_ptr }
    while (entry_strs[str_ent_ptr][ex_buf_ptr] ≠ end_of_string) do  { copy characters into the buffer }
      append_ex_buf_char(entry_strs[str_ent_ptr][ex_buf_ptr]);
    ex_buf_length ← ex_buf_ptr; add_pool_buf_and_push;  { push this string onto the stack }
    end;
  end
```

This code is used in section 325.

330. This module pushes the string given by a *str_global_var* onto the literal stack, but it copies the string to *str_pool* (character by character) only if it has to—it *doesn't* have to if the string is static (that is, if the string isn't at the top, temporary part of the string pool).

```
⟨ Execute a str_global_var 330 ⟩ ≡
  begin str_glb_ptr ← fn_info[ex_fn_loc];
  if (glb_str_ptr[str_glb_ptr] > 0) then { we're dealing with a static string }
    push_lit_stk(glb_str_ptr[str_glb_ptr], stk_str)
  else begin str_room(glb_str_end[str_glb_ptr]); glob_chr_ptr ← 0;
    while (glob_chr_ptr < glb_str_end[str_glb_ptr]) do { copy the string }
      begin append_char(global_strs[str_glb_ptr][glob_chr_ptr]); incr(glob_chr_ptr);
      end;
    push_lit_stk(make_string, stk_str); { and push it onto the stack }
    end;
  end
```

This code is used in section 325.

331. The built-in functions. This section gives the all the code for all the built-in functions (including pre-defined *field*s, *str_entry_var*s, and *int_global_var*s, which technically aren't classified as *built_in*). To modify or add one, we needn't go anywhere else (with one exception: The constant *max_pop*, which gives the maximum number of literals that any of these functions pops off the stack, is defined earlier because it's needed earlier; thus, if we need to update it, which will happen if some new *built_in* functions uses more than *max_pop* literals from the stack, we'll have to go outside this section). Adding a *built_in* function entails modifying (at least four of) the five modules marked by "add a built-in function" in the index, in addition to adding the code to execute the function.

These variables all begin with *b_* and specify the hash-table locations of the *built_in* functions, except that *b_default* is pseudo-*built_in*—either it will point to the no-op `skip$` or to the `.bst`-defined function `default.type`; it's used when an entry has a type that's not defined in the `.bst` file.

⟨ Globals in the outer block 16 ⟩ +≡
b_equals: *hash_loc*; { `=` }
b_greater_than: *hash_loc*; { `>` }
b_less_than: *hash_loc*; { `<` }
b_plus: *hash_loc*; { `+` (this may be changed to an *a_minus*) }
b_minus: *hash_loc*; { `-` }
b_concatenate: *hash_loc*; { `*` }
b_gets: *hash_loc*; { `:=` (formerly, *b_gat*) }
b_add_period: *hash_loc*; { `add.period$` }
b_call_type: *hash_loc*; { `call.type$` }
b_change_case: *hash_loc*; { `change.case$` }
b_chr_to_int: *hash_loc*; { `chr.to.int$` }
b_cite: *hash_loc*; { `cite$` }
b_duplicate: *hash_loc*; { `duplicate$` }
b_empty: *hash_loc*; { `empty$` }
b_format_name: *hash_loc*; { `format.name$` }
b_if: *hash_loc*; { `if$` }
b_int_to_chr: *hash_loc*; { `int.to.chr$` }
b_int_to_str: *hash_loc*; { `int.to.str$` }
b_missing: *hash_loc*; { `missing$` }
b_newline: *hash_loc*; { `newline$` }
b_num_names: *hash_loc*; { `num.names$` }
b_pop: *hash_loc*; { `pop$` }
b_preamble: *hash_loc*; { `preamble$` }
b_purify: *hash_loc*; { `purify$` }
b_quote: *hash_loc*; { `quote$` }
b_skip: *hash_loc*; { `skip$` }
b_stack: *hash_loc*; { `stack$` }
b_substring: *hash_loc*; { `substring$` }
b_swap: *hash_loc*; { `swap$` }
b_text_length: *hash_loc*; { `text.length$` }
b_text_prefix: *hash_loc*; { `text.prefix$` }
b_top_stack: *hash_loc*; { `top$` }
b_type: *hash_loc*; { `type$` }
b_warning: *hash_loc*; { `warning$` }
b_while: *hash_loc*; { `while$` }
b_width: *hash_loc*; { `width$` }
b_write: *hash_loc*; { `write$` }
b_default: *hash_loc*; { either `skip$` or `default.type` }

stat *blt_in_loc*: **array** [*blt_in_range*] **of** *hash_loc*; { for execution counts }
execution_count: **array** [*blt_in_range*] **of** *integer*; { the same }

total_ex_count: *integer*; { the sum of all *execution_count*s }
blt_in_ptr: *blt_in_range*; { a pointer into *blt_in_loc* }
tats

332. Where *blt_in_range* gives the legal *built_in* function numbers.

⟨ Types in the outer block 22 ⟩ +≡
blt_in_range = 0 .. *num_blt_in_fns*;

333. These constants all begin with *n_* and are used for the **case** statement that determines which *built_in* function to execute.

define *n_equals* = 0 { `=` }
define *n_greater_than* = 1 { `>` }
define *n_less_than* = 2 { `<` }
define *n_plus* = 3 { `+` }
define *n_minus* = 4 { `-` }
define *n_concatenate* = 5 { `*` }
define *n_gets* = 6 { `:=` }
define *n_add_period* = 7 { `add.period$` }
define *n_call_type* = 8 { `call.type$` }
define *n_change_case* = 9 { `change.case$` }
define *n_chr_to_int* = 10 { `chr.to.int$` }
define *n_cite* = 11 { `cite$` (this may start a riot) }
define *n_duplicate* = 12 { `duplicate$` }
define *n_empty* = 13 { `empty$` }
define *n_format_name* = 14 { `format.name$` }
define *n_if* = 15 { `if$` }
define *n_int_to_chr* = 16 { `int.to.chr$` }
define *n_int_to_str* = 17 { `int.to.str$` }
define *n_missing* = 18 { `missing$` }
define *n_newline* = 19 { `newline$` }
define *n_num_names* = 20 { `num.names$` }
define *n_pop* = 21 { `pop$` }
define *n_preamble* = 22 { `preamble$` }
define *n_purify* = 23 { `purify$` }
define *n_quote* = 24 { `quote$` }
define *n_skip* = 25 { `skip$` }
define *n_stack* = 26 { `stack$` }
define *n_substring* = 27 { `substring$` }
define *n_swap* = 28 { `swap$` }
define *n_text_length* = 29 { `text.length$` }
define *n_text_prefix* = 30 { `text.prefix$` }
define *n_top_stack* = 31 { `top$` }
define *n_type* = 32 { `type$` }
define *n_warning* = 33 { `warning$` }
define *n_while* = 34 { `while$` }
define *n_width* = 35 { `width$` }
define *n_write* = 36 { `write$` }

⟨ Constants in the outer block 14 ⟩ +≡
num_blt_in_fns = 37; { one more than the previous number }

334. It's time for us to insert more pre-defined strings into *str_pool* (and thus the hash table) and to insert the *built_in* functions into the hash table. The strings corresponding to these functions should contain no upper-case letters, and they must all be exactly *longest_pds* characters long. The *build_in* routine (to appear shortly) does the work.

Important note: These pre-definitions must not have any glitches or the program may bomb because the *log_file* hasn't been opened yet.

⟨ Pre-define certain strings 75 ⟩ +≡
build_in(´=␣␣␣␣␣␣␣␣␣␣´, 1, *b_equals*, *n_equals*);
build_in(´>␣␣␣␣␣␣␣␣␣␣´, 1, *b_greater_than*, *n_greater_than*);
build_in(´<␣␣␣␣␣␣␣␣␣␣´, 1, *b_less_than*, *n_less_than*); *build_in*(´+␣␣␣␣␣␣␣␣␣␣´, 1, *b_plus*, *n_plus*);
build_in(´-␣␣␣␣␣␣␣␣␣␣´, 1, *b_minus*, *n_minus*);
build_in(´*␣␣␣␣␣␣␣␣␣␣´, 1, *b_concatenate*, *n_concatenate*); *build_in*(´:=␣␣␣␣␣␣␣␣␣´, 2, *b_gets*, *n_gets*);
build_in(´add.period$␣´, 11, *b_add_period*, *n_add_period*);
build_in(´call.type$␣␣´, 10, *b_call_type*, *n_call_type*);
build_in(´change.case$´, 12, *b_change_case*, *n_change_case*);
build_in(´chr.to.int$␣´, 11, *b_chr_to_int*, *n_chr_to_int*); *build_in*(´cite$␣␣␣␣␣␣␣´, 5, *b_cite*, *n_cite*);
build_in(´duplicate$␣␣´, 10, *b_duplicate*, *n_duplicate*); *build_in*(´empty$␣␣␣␣␣␣´, 6, *b_empty*, *n_empty*);
build_in(´format.name$´, 12, *b_format_name*, *n_format_name*); *build_in*(´if$␣␣␣␣␣␣␣␣␣´, 3, *b_if*, *n_if*);
build_in(´int.to.chr$␣´, 11, *b_int_to_chr*, *n_int_to_chr*);
build_in(´int.to.str$␣´, 11, *b_int_to_str*, *n_int_to_str*);
build_in(´missing$␣␣␣␣´, 8, *b_missing*, *n_missing*); *build_in*(´newline$␣␣␣␣´, 8, *b_newline*, *n_newline*);
build_in(´num.names$␣␣´, 10, *b_num_names*, *n_num_names*); *build_in*(´pop$␣␣␣␣␣␣␣␣´, 4, *b_pop*, *n_pop*);
build_in(´preamble$␣␣␣´, 9, *b_preamble*, *n_preamble*); *build_in*(´purify$␣␣␣␣␣´, 7, *b_purify*, *n_purify*);
build_in(´quote$␣␣␣␣␣␣´, 6, *b_quote*, *n_quote*); *build_in*(´skip$␣␣␣␣␣␣␣´, 5, *b_skip*, *n_skip*);
build_in(´stack$␣␣␣␣␣␣´, 6, *b_stack*, *n_stack*); *build_in*(´substring$␣␣´, 10, *b_substring*, *n_substring*);
build_in(´swap$␣␣␣␣␣␣␣´, 5, *b_swap*, *n_swap*); *build_in*(´text.length$´, 12, *b_text_length*, *n_text_length*);
build_in(´text.prefix$´, 12, *b_text_prefix*, *n_text_prefix*);
build_in(´top$␣␣␣␣␣␣␣␣´, 4, *b_top_stack*, *n_top_stack*); *build_in*(´type$␣␣␣␣␣␣␣´, 5, *b_type*, *n_type*);
build_in(´warning$␣␣␣␣´, 8, *b_warning*, *n_warning*); *build_in*(´width$␣␣␣␣␣´, 6, *b_width*, *n_width*);
build_in(´while$␣␣␣␣␣␣´, 6, *b_while*, *n_while*); *build_in*(´width$␣␣␣␣␣´, 6, *b_width*, *n_width*);
build_in(´write$␣␣␣␣␣␣´, 6, *b_write*, *n_write*);

335. This procedure inserts a *built_in* function into the hash table and initializes the corresponding pre-defined string (of length at most *longest_pds*). The array *fn_info* contains a number from 0 through the number of *built_in* functions minus 1 (i.e., *num_blt_in_fns* − 1 if we're keeping statistics); this number is used by a **case** statement to execute this function and is used for keeping execution counts when keeping statistics.

⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure *build_in*(*pds* : *pds_type*; *len* : *pds_len*; **var** *fn_hash_loc* : *hash_loc*; *blt_in_num* : *blt_in_range*);
begin *pre_define*(*pds*, *len*, *bst_fn_ilk*);
fn_hash_loc ← *pre_def_loc*; { the *pre_define* routine sets *pre_def_loc* }
fn_type[*fn_hash_loc*] ← *built_in*; *fn_info*[*fn_hash_loc*] ← *blt_in_num*;
stat *blt_in_loc*[*blt_in_num*] ← *fn_hash_loc*;
execution_count[*blt_in_num*] ← 0; { initialize the function-execution count }
tats
end;

336. This is a procedure so that *initialize* is smaller.

⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure *pre_def_certain_strings*;
  begin ⟨ Pre-define certain strings 75 ⟩
  end;

337. These variables all begin with *s_* and specify the locations in *str_pool* of certain often-used strings that the `.bst` commands need. The *s_preamble* array is big enough to allow an average of one `preamble$` command per `.bib` file.

⟨ Globals in the outer block 16 ⟩ +≡
s_null: *str_number*; { the null string }
s_default: *str_number*; { `default.type`, for unknown entry types }
s_t: *str_number*; { `t`, for *title_lowers* case conversion }
s_l: *str_number*; { `l`, for *all_lowers* case conversion }
s_u: *str_number*; { `u`, for *all_uppers* case conversion }
s_preamble: **array** [*bib_number*] **of** *str_number*; { for the `preamble$` *built_in* function }

338. These constants all begin with *n_* and are used for the **case** statement that determines which, if any, control sequence we're dealing with; a control sequence of interest will be either one of the undotted characters '`\i`' or '`\j`' or one of the foreign characters in Table 3.2 of the LaTeX manual.

  define *n_i* = 0 { `i`, for the undotted character `\i` }
  define *n_j* = 1 { `j`, for the undotted character `\j` }
  define *n_oe* = 2 { `oe`, for the foreign character `\oe` }
  define *n_oe_upper* = 3 { `OE`, for the foreign character `\OE` }
  define *n_ae* = 4 { `ae`, for the foreign character `\ae` }
  define *n_ae_upper* = 5 { `AE`, for the foreign character `\AE` }
  define *n_aa* = 6 { `aa`, for the foreign character `\aa` }
  define *n_aa_upper* = 7 { `AA`, for the foreign character `\AA` }
  define *n_o* = 8 { `o`, for the foreign character `\o` }
  define *n_o_upper* = 9 { `O`, for the foreign character `\O` }
  define *n_l* = 10 { `l`, for the foreign character `\l` }
  define *n_l_upper* = 11 { `L`, for the foreign character `\L` }
  define *n_ss* = 12 { `ss`, for the foreign character `\ss` }

339. Here we pre-define a few strings used in executing the `.bst` file: the null string, which is sometimes pushed onto the stack; a string used for default entry types; and some control sequences used to spot foreign characters. We also initialize the *s_preamble* array to empty. These pre-defined strings must all be exactly *longest_pds* characters long.

Important note: These pre-definitions must not have any glitches or the program may bomb because the *log_file* hasn't been opened yet, and *text_ilk*s should be pre-defined here, not earlier, for `.bst`-function-execution purposes.

⟨ Pre-define certain strings 75 ⟩ +≡
pre_define(´␣␣␣␣␣␣␣␣␣␣␣␣␣´, 0, *text_ilk*); *s_null* ← *hash_text*[*pre_def_loc*]; *fn_type*[*pre_def_loc*] ← *str_literal*;
pre_define(´`default.type`´, 12, *text_ilk*); *s_default* ← *hash_text*[*pre_def_loc*];
fn_type[*pre_def_loc*] ← *str_literal*;
b_default ← *b_skip*; { this may be changed to the `default.type` function }
preamble_ptr ← 0; { initialize the *s_preamble* array }
pre_define(´`i`␣␣␣␣␣␣␣␣␣␣␣´, 1, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_i*;
pre_define(´`j`␣␣␣␣␣␣␣␣␣␣␣´, 1, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_j*;
pre_define(´`oe`␣␣␣␣␣␣␣␣␣␣´, 2, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_oe*;
pre_define(´`OE`␣␣␣␣␣␣␣␣␣␣´, 2, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_oe_upper*;
pre_define(´`ae`␣␣␣␣␣␣␣␣␣␣´, 2, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_ae*;
pre_define(´`AE`␣␣␣␣␣␣␣␣␣␣´, 2, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_ae_upper*;
pre_define(´`aa`␣␣␣␣␣␣␣␣␣␣´, 2, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_aa*;
pre_define(´`AA`␣␣␣␣␣␣␣␣␣␣´, 2, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_aa_upper*;
pre_define(´`o`␣␣␣␣␣␣␣␣␣␣␣´, 1, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_o*;
pre_define(´`O`␣␣␣␣␣␣␣␣␣␣␣´, 1, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_o_upper*;
pre_define(´`l`␣␣␣␣␣␣␣␣␣␣␣´, 1, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_l*;
pre_define(´`L`␣␣␣␣␣␣␣␣␣␣␣´, 1, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_l_upper*;
pre_define(´`ss`␣␣␣␣␣␣␣␣␣␣´, 2, *control_seq_ilk*); *ilk_info*[*pre_def_loc*] ← *n_ss*;

340. Now we pre-define any built-in *field*s, *str_entry_var*s, and *int_global_var*s; these strings must all be exactly *longest_pds* characters long. Note that although these are built-in functions, we classify them (in the *fn_type* array) otherwise.

Important note: These pre-definitions must not have any glitches or the program may bomb because the *log_file* hasn't been opened yet.

⟨ Pre-define certain strings 75 ⟩ +≡
pre_define(´`crossref`␣␣␣␣´, 8, *bst_fn_ilk*); *fn_type*[*pre_def_loc*] ← *field*;
fn_info[*pre_def_loc*] ← *num_fields*; { give this *field* a number }
crossref_num ← *num_fields*; *incr*(*num_fields*);
num_pre_defined_fields ← *num_fields*; { that's it for pre-defined *field*s }
pre_define(´`sort.key$`␣␣␣´, 9, *bst_fn_ilk*); *fn_type*[*pre_def_loc*] ← *str_entry_var*;
fn_info[*pre_def_loc*] ← *num_ent_strs*; { give this *str_entry_var* a number }
sort_key_num ← *num_ent_strs*; *incr*(*num_ent_strs*);
pre_define(´`entry.max$`␣␣´, 10, *bst_fn_ilk*); *fn_type*[*pre_def_loc*] ← *int_global_var*;
fn_info[*pre_def_loc*] ← *ent_str_size*; { initialize this *int_global_var* }
pre_define(´`global.max$`␣´, 11, *bst_fn_ilk*); *fn_type*[*pre_def_loc*] ← *int_global_var*;
fn_info[*pre_def_loc*] ← *glob_str_size*; { initialize this *int_global_var* }

341. This module branches to the code for the appropriate *built_in* function. Only three—`call.type$`, `if$`, and `while$`—do a recursive call.

⟨ Execute a *built_in* function 341 ⟩ ≡
```
begin stat   { update this function's execution count }
incr(execution_count[fn_info[ex_fn_loc]]);
tats
case (fn_info[ex_fn_loc]) of
n_equals: x_equals;
n_greater_than: x_greater_than;
n_less_than: x_less_than;
n_plus: x_plus;
n_minus: x_minus;
n_concatenate: x_concatenate;
n_gets: x_gets;
n_add_period: x_add_period;
n_call_type: ⟨ execute_fn(call.type$) 363 ⟩;
n_change_case: x_change_case;
n_chr_to_int: x_chr_to_int;
n_cite: x_cite;
n_duplicate: x_duplicate;
n_empty: x_empty;
n_format_name: x_format_name;
n_if: ⟨ execute_fn(if$) 421 ⟩;
n_int_to_chr: x_int_to_chr;
n_int_to_str: x_int_to_str;
n_missing: x_missing;
n_newline: ⟨ execute_fn(newline$) 425 ⟩;
n_num_names: x_num_names;
n_pop: ⟨ execute_fn(pop$) 428 ⟩;
n_preamble: x_preamble;
n_purify: x_purify;
n_quote: x_quote;
n_skip: ⟨ execute_fn(skip$) 435 ⟩;
n_stack: ⟨ execute_fn(stack$) 436 ⟩;
n_substring: x_substring;
n_swap: x_swap;
n_text_length: x_text_length;
n_text_prefix: x_text_prefix;
n_top_stack: ⟨ execute_fn(top$) 446 ⟩;
n_type: x_type;
n_warning: x_warning;
n_while: ⟨ execute_fn(while$) 449 ⟩;
n_width: x_width;
n_write: x_write;
othercases confusion('Unknown␣built-in␣function')
endcases;
end
```
This code is used in section 325.

342. This extra level of module-pointing allows a uniformity of module names for the *built_in* functions, regardless of whether they do a recursive call to *execute_fn* or are trivial (a single statement). Those that do a recursive call are left as part of *execute_fn*, avoiding PASCAL's forward procedure mechanism, and those that don't (except for the single-statement ones) are made into procedures so that *execute_fn* doesn't get too large.

⟨ Procedures and functions for style-file function execution 307 ⟩ +≡
⟨ *execute_fn*(=) 345 ⟩
⟨ *execute_fn*(>) 346 ⟩
⟨ *execute_fn*(<) 347 ⟩
⟨ *execute_fn*(+) 348 ⟩
⟨ *execute_fn*(-) 349 ⟩
⟨ *execute_fn*(*) 350 ⟩
⟨ *execute_fn*(:=) 354 ⟩
⟨ *execute_fn*(add.period$) 360 ⟩
⟨ *execute_fn*(change.case$) 364 ⟩
⟨ *execute_fn*(chr.to.int$) 377 ⟩
⟨ *execute_fn*(cite$) 378 ⟩
⟨ *execute_fn*(duplicate$) 379 ⟩
⟨ *execute_fn*(empty$) 380 ⟩
⟨ *execute_fn*(format.name$) 382 ⟩
⟨ *execute_fn*(int.to.chr$) 422 ⟩
⟨ *execute_fn*(int.to.str$) 423 ⟩
⟨ *execute_fn*(missing$) 424 ⟩
⟨ *execute_fn*(num.names$) 426 ⟩
⟨ *execute_fn*(preamble$) 429 ⟩
⟨ *execute_fn*(purify$) 430 ⟩
⟨ *execute_fn*(quote$) 434 ⟩
⟨ *execute_fn*(substring$) 437 ⟩
⟨ *execute_fn*(swap$) 439 ⟩
⟨ *execute_fn*(text.length$) 441 ⟩
⟨ *execute_fn*(text.prefix$) 443 ⟩
⟨ *execute_fn*(type$) 447 ⟩
⟨ *execute_fn*(warning$) 448 ⟩
⟨ *execute_fn*(width$) 450 ⟩
⟨ *execute_fn*(write$) 454 ⟩
⟨ *execute_fn* itself 325 ⟩

343. Now it's time to declare some things for executing *built_in* functions only. These (and only these) variables are used recursively, so they can't be global.

define *end_while* = 51 { stop executing the `while$` function }

⟨ Declarations for executing *built_in* functions 343 ⟩ ≡
label *end_while*;
var *r_pop_lt1*, *r_pop_lt2*: *integer*; { stack literals for `while$` }
r_pop_tp1, *r_pop_tp2*: *stk_type*; { stack types for `while$` }
This code is used in section 325.

344. These are nonrecursive variables that *execute_fn* uses. Declaring them here (instead of in the previous module) saves execution time and stack space on most machines.

define *name_buf* ≡ *sv_buffer* { an alias, a buffer for manipulating names }

⟨ Globals in the outer block 16 ⟩ +≡
pop_lit1, *pop_lit2*, *pop_lit3*: *integer*; { stack literals }
pop_typ1, *pop_typ2*, *pop_typ3*: *stk_type*; { stack types }
sp_ptr: *pool_pointer*; { for manipulating *str_pool* strings }
sp_xptr1, *sp_xptr2*: *pool_pointer*; { more of the same }
sp_end: *pool_pointer*; { marks the end of a *str_pool* string }
sp_length, *sp2_length*: *pool_pointer*; { lengths of *str_pool* strings }
sp_brace_level: *integer*; { for scanning *str_pool* strings }
ex_buf_xptr, *ex_buf_yptr*: *buf_pointer*; { extra *ex_buf* locations }
control_seq_loc: *hash_loc*; { hash-table loc of a control sequence }
preceding_white: *boolean*; { used in scanning strings }
and_found: *boolean*; { to stop the loop that looks for an "and" }
num_names: *integer*; { for counting names }
name_bf_ptr: *buf_pointer*; { general *name_buf* location }
name_bf_xptr, *name_bf_yptr*: *buf_pointer*; { and two more }
nm_brace_level: *integer*; { for scanning *name_buf* strings }
name_tok: **packed array** [*buf_pointer*] **of** *buf_pointer*; { name-token ptr list }
name_sep_char: **packed array** [*buf_pointer*] **of** *ASCII_code*; { token-ending chars }
num_tokens: *buf_pointer*; { this counts name tokens }
token_starting: *boolean*; { used in scanning name tokens }
alpha_found: *boolean*; { used in scanning the format string }
double_letter, *end_of_group*, *to_be_written*: *boolean*; { the same }
first_start: *buf_pointer*; { start-ptr into *name_tok* for the first name }
first_end: *buf_pointer*; { end-ptr into *name_tok* for the first name }
last_end: *buf_pointer*; { end-ptr into *name_tok* for the last name }
von_start: *buf_pointer*; { start-ptr into *name_tok* for the von name }
von_end: *buf_pointer*; { end-ptr into *name_tok* for the von name }
jr_end: *buf_pointer*; { end-ptr into *name_tok* for the jr name }
cur_token, *last_token*: *buf_pointer*; { *name_tok* ptrs for outputting tokens }
use_default: *boolean*; { for the inter-token intra-name part string }
num_commas: *buf_pointer*; { used to determine the name syntax }
comma1, *comma2*: *buf_pointer*; { ptrs into *name_tok* }
num_text_chars: *buf_pointer*; { special characters count as one }

345. The *built_in* function `=` pops the top two (integer or string) literals, compares them, and pushes the integer 1 if they're equal, 0 otherwise. If they're not either both string or both integer, it complains and pushes the integer 0.

```
⟨ execute_fn(=) 345 ⟩ ≡
procedure x_equals;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ pop_typ2) then
    begin if ((pop_typ1 ≠ stk_empty) ∧ (pop_typ2 ≠ stk_empty)) then
      begin print_stk_lit(pop_lit1, pop_typ1); print(´,␣´); print_stk_lit(pop_lit2, pop_typ2); print_newline;
      bst_ex_warn(´---they␣aren´´t␣the␣same␣literal␣types´);
      end;
    push_lit_stk(0, stk_int);
    end
  else if ((pop_typ1 ≠ stk_int) ∧ (pop_typ1 ≠ stk_str)) then
      begin if (pop_typ1 ≠ stk_empty) then
        begin print_stk_lit(pop_lit1, pop_typ1); bst_ex_warn(´,␣not␣an␣integer␣or␣a␣string,´);
        end;
      push_lit_stk(0, stk_int);
      end
    else if (pop_typ1 = stk_int) then
        if (pop_lit2 = pop_lit1) then push_lit_stk(1, stk_int)
        else push_lit_stk(0, stk_int)
      else if (str_eq_str(pop_lit2, pop_lit1)) then push_lit_stk(1, stk_int)
        else push_lit_stk(0, stk_int);
  end;
```

This code is used in section 342.

346. The *built_in* function `>` pops the top two (integer) literals, compares them, and pushes the integer 1 if the second is greater than the first, 0 otherwise. If either isn't an integer literal, it complains and pushes the integer 0.

```
⟨ execute_fn(>) 346 ⟩ ≡
procedure x_greater_than;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(0, stk_int);
    end
  else if (pop_typ2 ≠ stk_int) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int); push_lit_stk(0, stk_int);
      end
    else if (pop_lit2 > pop_lit1) then push_lit_stk(1, stk_int)
      else push_lit_stk(0, stk_int);
  end;
```

This code is used in section 342.

347. The *built_in* function `<` pops the top two (integer) literals, compares them, and pushes the integer 1 if the second is less than the first, 0 otherwise. If either isn't an integer literal, it complains and pushes the integer 0.

```
⟨ execute_fn(<) 347 ⟩ ≡
procedure x_less_than;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(0, stk_int);
    end
  else if (pop_typ2 ≠ stk_int) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int); push_lit_stk(0, stk_int);
      end
    else if (pop_lit2 < pop_lit1) then push_lit_stk(1, stk_int)
      else push_lit_stk(0, stk_int);
  end;
```

This code is used in section 342.

348. The *built_in* function `+` pops the top two (integer) literals and pushes their sum. If either isn't an integer literal, it complains and pushes the integer 0.

```
⟨ execute_fn(+) 348 ⟩ ≡
procedure x_plus;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(0, stk_int);
    end
  else if (pop_typ2 ≠ stk_int) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int); push_lit_stk(0, stk_int);
      end
    else push_lit_stk(pop_lit2 + pop_lit1, stk_int);
  end;
```

This code is used in section 342.

349. The *built_in* function `-` pops the top two (integer) literals and pushes their difference (the first subtracted from the second). If either isn't an integer literal, it complains and pushes the integer 0.

```
⟨ execute_fn(-) 349 ⟩ ≡
procedure x_minus;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(0, stk_int);
    end
  else if (pop_typ2 ≠ stk_int) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int); push_lit_stk(0, stk_int);
      end
    else push_lit_stk(pop_lit2 − pop_lit1, stk_int);
  end;
```

This code is used in section 342.

350. The *built_in* function * pops the top two (string) literals, concatenates them (in reverse order, that is, the order in which pushed), and pushes the resulting string back onto the stack. If either isn't a string literal, it complains and pushes the null string.

```
⟨ execute_fn(*) 350 ⟩ ≡
procedure x_concatenate;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(s_null, stk_str);
    end
  else if (pop_typ2 ≠ stk_str) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_str); push_lit_stk(s_null, stk_str);
      end
    else ⟨ Concatenate the two strings and push 351 ⟩;
  end;
```

This code is used in section 342.

351. Often both strings will be at the top of the string pool, in which case we just move some pointers. Furthermore, it's worth doing some special stuff in case either string is null, since empirically this seems to happen about 20% of the time. In any case, we don't need the execution buffer—we simple move the strings around in the string pool when necessary.

```
⟨ Concatenate the two strings and push 351 ⟩ ≡
  begin if (pop_lit2 ≥ cmd_str_ptr) then
    if (pop_lit1 ≥ cmd_str_ptr) then
      begin str_start[pop_lit1] ← str_start[pop_lit1 + 1]; unflush_string; incr(lit_stk_ptr);
      end
    else if (length(pop_lit2) = 0) then push_lit_stk(pop_lit1, stk_str)
      else   { pop_lit2 is nonnull, only pop_lit1 is below cmd_str_ptr }
  begin pool_ptr ← str_start[pop_lit2 + 1]; str_room(length(pop_lit1)); sp_ptr ← str_start[pop_lit1];
  sp_end ← str_start[pop_lit1 + 1];
  while (sp_ptr < sp_end) do
    begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
    end;
  push_lit_stk(make_string, stk_str);   { and push it onto the stack }
  end
else ⟨ Concatenate them and push when pop_lit2 < cmd_str_ptr 352 ⟩;
  end
```

This code is used in section 350.

352. We simply continue the previous module.

```
⟨ Concatenate them and push when pop_lit2 < cmd_str_ptr 352 ⟩ ≡
  begin if (pop_lit1 ≥ cmd_str_ptr) then
    if (length(pop_lit2) = 0) then
      begin unflush_string; lit_stack[lit_stk_ptr] ← pop_lit1; incr(lit_stk_ptr);
      end
    else if (length(pop_lit1) = 0) then incr(lit_stk_ptr)
      else { both strings nonnull, only pop_lit2 is below cmd_str_ptr }
  begin sp_length ← length(pop_lit1); sp2_length ← length(pop_lit2); str_room(sp_length + sp2_length);
  sp_ptr ← str_start[pop_lit1 + 1]; sp_end ← str_start[pop_lit1]; sp_xptr1 ← sp_ptr + sp2_length;
  while (sp_ptr > sp_end) do { slide up pop_lit1 }
    begin decr(sp_ptr); decr(sp_xptr1); str_pool[sp_xptr1] ← str_pool[sp_ptr];
    end;
  sp_ptr ← str_start[pop_lit2]; sp_end ← str_start[pop_lit2 + 1];
  while (sp_ptr < sp_end) do { slide up pop_lit2 }
    begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
    end;
  pool_ptr ← pool_ptr + sp_length; push_lit_stk(make_string, stk_str); { and push it onto the stack }
  end
else ⟨ Concatenate them and push when pop_lit1, pop_lit2 < cmd_str_ptr 353 ⟩;
  end
```

This code is used in section 351.

353. Again, we simply continue the previous module.

```
⟨ Concatenate them and push when pop_lit1, pop_lit2 < cmd_str_ptr 353 ⟩ ≡
  begin if (length(pop_lit1) = 0) then incr(lit_stk_ptr)
  else if (length(pop_lit2) = 0) then push_lit_stk(pop_lit1, stk_str)
    else { both strings are nonnull, and both are below cmd_str_ptr }
  begin str_room(length(pop_lit1) + length(pop_lit2)); sp_ptr ← str_start[pop_lit2];
  sp_end ← str_start[pop_lit2 + 1];
  while (sp_ptr < sp_end) do { slide up pop_lit2 }
    begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
    end;
  sp_ptr ← str_start[pop_lit1]; sp_end ← str_start[pop_lit1 + 1];
  while (sp_ptr < sp_end) do { slide up pop_lit1 }
    begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
    end;
  push_lit_stk(make_string, stk_str); { and push it onto the stack }
  end;
  end
```

This code is used in section 352.

354. The *built_in* function `:=` pops the top two literals and assigns to the first (which must be an *int_entry_var*, a *str_entry_var*, an *int_global_var*, or a *str_global_var*) the value of the second; it complains if the value isn't of the appropriate type.

```
⟨ execute_fn(:=) 354 ⟩ ≡
procedure x_gets;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_fn) then print_wrong_stk_lit(pop_lit1, pop_typ1, stk_fn)
  else if ((¬mess_with_entries) ∧ ((fn_type[pop_lit1] = str_entry_var) ∨ (fn_type[pop_lit1] = int_entry_var)))
          then bst_cant_mess_with_entries_print
    else case (fn_type[pop_lit1]) of
      int_entry_var: ⟨ Assign to an int_entry_var 355 ⟩;
      str_entry_var: ⟨ Assign to a str_entry_var 357 ⟩;
      int_global_var: ⟨ Assign to an int_global_var 358 ⟩;
      str_global_var: ⟨ Assign to a str_global_var 359 ⟩;
      othercases begin print('You␣can''t␣assign␣to␣type␣'); print_fn_class(pop_lit1);
        bst_ex_warn(',␣a␣nonvariable␣function␣class');
        end
      endcases;
  end;
```

This code is used in section 342.

355. This module checks that what we're about to assign is really an integer, and then assigns.

```
⟨ Assign to an int_entry_var 355 ⟩ ≡
  if (pop_typ2 ≠ stk_int) then print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int)
  else entry_ints[cite_ptr * num_ent_ints + fn_info[pop_lit1]] ← pop_lit2
```

This code is used in section 354.

356. It's time for a complaint if either of the two (entry or global) string lengths is exceeded.

```
  define bst_string_size_exceeded(#) ≡
           begin bst_1print_string_size_exceeded; print(#); bst_2print_string_size_exceeded;
           end

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure bst_1print_string_size_exceeded;
  begin print('Warning--you''ve␣exceeded␣');
  end;

procedure bst_2print_string_size_exceeded;
  begin print('-string-size,'); bst_mild_ex_warn_print;
  print_ln('*Please␣notify␣the␣bibstyle␣designer*');
  end;
```

357. This module checks that what we're about to assign is really a string, and then assigns.

```
⟨ Assign to a str_entry_var 357 ⟩ ≡
  begin if (pop_typ2 ≠ stk_str) then print_wrong_stk_lit(pop_lit2, pop_typ2, stk_str)
  else begin str_ent_ptr ← cite_ptr * num_ent_strs + fn_info[pop_lit1]; ent_chr_ptr ← 0;
    sp_ptr ← str_start[pop_lit2]; sp_xptr1 ← str_start[pop_lit2 + 1];
    if (sp_xptr1 − sp_ptr > ent_str_size) then
      begin bst_string_size_exceeded(ent_str_size : 0, ´,␣the␣entry´); sp_xptr1 ← sp_ptr + ent_str_size;
      end;
    while (sp_ptr < sp_xptr1) do
      begin { copy characters into entry_strs }
      entry_strs[str_ent_ptr][ent_chr_ptr] ← str_pool[sp_ptr]; incr(ent_chr_ptr); incr(sp_ptr);
      end;
    entry_strs[str_ent_ptr][ent_chr_ptr] ← end_of_string;
    end
  end
```

This code is used in section 354.

358. This module checks that what we're about to assign is really an integer, and then assigns.

```
⟨ Assign to an int_global_var 358 ⟩ ≡
  if (pop_typ2 ≠ stk_int) then print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int)
  else fn_info[pop_lit1] ← pop_lit2
```

This code is used in section 354.

359. This module checks that what we're about to assign is really a string, and then assigns.

```
⟨ Assign to a str_global_var 359 ⟩ ≡
  begin if (pop_typ2 ≠ stk_str) then print_wrong_stk_lit(pop_lit2, pop_typ2, stk_str)
  else begin str_glb_ptr ← fn_info[pop_lit1];
    if (pop_lit2 < cmd_str_ptr) then glb_str_ptr[str_glb_ptr] ← pop_lit2
    else begin glb_str_ptr[str_glb_ptr] ← 0; glob_chr_ptr ← 0; sp_ptr ← str_start[pop_lit2];
      sp_end ← str_start[pop_lit2 + 1];
      if (sp_end − sp_ptr > glob_str_size) then
        begin bst_string_size_exceeded(glob_str_size : 0, ´,␣the␣global´); sp_end ← sp_ptr + glob_str_size;
        end;
      while (sp_ptr < sp_end) do
        begin { copy characters into global_strs }
        global_strs[str_glb_ptr][glob_chr_ptr] ← str_pool[sp_ptr]; incr(glob_chr_ptr); incr(sp_ptr);
        end;
      glb_str_end[str_glb_ptr] ← glob_chr_ptr;
      end;
    end
  end
```

This code is used in section 354.

360. The *built_in* function `add.period$` pops the top (string) literal, adds a *period* to a nonnull string if its last non*right_brace* character isn't a *period*, *question_mark*, or *exclamation_mark*, and pushes this resulting string back onto the stack. If the literal isn't a string, it complains and pushes the null string.

```
⟨ execute_fn(add.period$) 360 ⟩ ≡
procedure x_add_period;
  label loop_exit;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(s_null, stk_str);
    end
  else if (length(pop_lit1) = 0) then   { don't add period to the null string }
      push_lit_stk(s_null, stk_str)
    else ⟨ Add the period, if necessary, and push 361 ⟩;
  end;
```

This code is used in section 342.

361. Here we scan backwards from the end of the string, skipping non*right_brace* characters, to see if we have to add the *period*.

```
⟨ Add the period, if necessary, and push 361 ⟩ ≡
  begin sp_ptr ← str_start[pop_lit1 + 1]; sp_end ← str_start[pop_lit1];
  while (sp_ptr > sp_end) do   { find a nonright_brace }
    begin decr(sp_ptr);
    if (str_pool[sp_ptr] ≠ right_brace) then goto loop_exit;
    end;
loop_exit: case (str_pool[sp_ptr]) of
  period, question_mark, exclamation_mark: repush_string;
  othercases ⟨ Add the period (it's necessary) and push 362 ⟩
  endcases;
  end
```

This code is used in section 360.

362. Ok guys, we really have to do it.

```
⟨ Add the period (it's necessary) and push 362 ⟩ ≡
  begin if (pop_lit1 < cmd_str_ptr) then
    begin str_room(length(pop_lit1) + 1); sp_ptr ← str_start[pop_lit1]; sp_end ← str_start[pop_lit1 + 1];
    while (sp_ptr < sp_end) do   { slide pop_lit1 atop the string pool }
      begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
      end;
    end
  else   { the string is already there }
  begin pool_ptr ← str_start[pop_lit1 + 1]; str_room(1);
  end; append_char(period); push_lit_stk(make_string, stk_str);
  end
```

This code is used in section 361.

363. The *built_in* function `call.type$` executes the function specified in *type_list* for this entry unless it's *undefined*, in which case it executes the default function `default.type` defined in the `.bst` file, or unless it's *empty*, in which case it does nothing.

```
⟨ execute_fn(call.type$) 363 ⟩ ≡
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else if (type_list[cite_ptr] = undefined) then execute_fn(b_default)
    else if (type_list[cite_ptr] = empty) then do_nothing
      else execute_fn(type_list[cite_ptr]);
  end
```

This code is used in section 341.

364. The *built_in* function `change.case$` pops the top two (string) literals; it changes the case of the second according to the specifications of the first, as follows. (Note: The word 'letters' in the next sentence refers only to those at brace-level 0, the top-most brace level; no other characters are changed, except perhaps for special characters, described shortly.) If the first literal is the string `t`, it converts to lower case all letters except the very first character in the string, which it leaves alone, and except the first character following any *colon* and then nonnull *white_space*, which it also leaves alone; if it's the string `l`, it converts all letters to lower case; if it's the string `u`, it converts all letters to upper case; and if it's anything else, it complains and does no conversion. It then pushes this resulting string. If either type is incorrect, it complains and pushes the null string; however, if both types are correct but the specification string (i.e., the first string) isn't one of the legal ones, it merely pushes the second back onto the stack, after complaining. (Another note: It ignores case differences in the specification string; for example, the strings `t` and `T` are equivalent for the purposes of this *built_in* function.)

```
  define ok_pascal_i_give_up = 21
⟨ execute_fn(change.case$) 364 ⟩ ≡
procedure x_change_case;
  label ok_pascal_i_give_up;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(s_null, stk_str);
    end
  else if (pop_typ2 ≠ stk_str) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_str); push_lit_stk(s_null, stk_str);
      end
    else begin ⟨ Determine the case-conversion type 366 ⟩;
      ex_buf_length ← 0; add_buf_pool(pop_lit2); ⟨ Perform the case conversion 370 ⟩;
      add_pool_buf_and_push; { push this string onto the stack }
      end;
  end;
```

This code is used in section 342.

365. First we define a few variables for case conversion. The constant definitions, to be used in **case** statements, are in order of probable frequency.

```
  define title_lowers = 0 { representing the string t }
  define all_lowers = 1 { representing the string l }
  define all_uppers = 2 { representing the string u }
  define bad_conversion = 3 { representing any illegal case-conversion string }
⟨ Globals in the outer block 16 ⟩ +≡
conversion_type: 0 .. bad_conversion; { the possible cases }
prev_colon: boolean; { true if just past a colon }
```

366. Now we determine which of the three case-conversion types we're dealing with: `t`, `l`, or `u`.

```
⟨ Determine the case-conversion type 366 ⟩ ≡
  begin case (str_pool[str_start[pop_lit1]]) of
  "t","T": conversion_type ← title_lowers;
  "l","L": conversion_type ← all_lowers;
  "u","U": conversion_type ← all_uppers;
  othercases conversion_type ← bad_conversion
  endcases;
  if ((length(pop_lit1) ≠ 1) ∨ (conversion_type = bad_conversion)) then
    begin conversion_type ← bad_conversion; print_pool_str(pop_lit1);
    bst_ex_warn('␣is␣an␣illegal␣case-conversion␣string');
    end;
  end
```

This code is used in section 364.

367. This procedure complains if the just-encountered *right_brace* would make *brace_level* negative.

```
⟨ Procedures and functions for name-string processing 367 ⟩ ≡
procedure decr_brace_level(pop_lit_var : str_number);
  begin if (brace_level = 0) then braces_unbalanced_complaint(pop_lit_var)
  else decr(brace_level);
  end;
```

See also sections 369, 384, 397, 401, 404, 406, 418·$10^T s$420.

This code is used in section 12.

368. This complaint often arises because the style designer has to type lots of braces.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure braces_unbalanced_complaint(pop_lit_var : str_number);
  begin print('Warning--"'); print_pool_str(pop_lit_var);
  bst_mild_ex_warn('"␣isn''t␣a␣brace-balanced␣string');
  end;
```

369. This one makes sure that *brace_level* = 0 (it's called at a point in a string where braces must be balanced).

```
⟨ Procedures and functions for name-string processing 367 ⟩ +≡
procedure check_brace_level(pop_lit_var : str_number);
  begin if (brace_level > 0) then braces_unbalanced_complaint(pop_lit_var);
  end;
```

370. Here's where we actually go through the string and do the case conversion.

```
⟨ Perform the case conversion 370 ⟩ ≡
  begin brace_level ← 0;  { this is the top level }
  ex_buf_ptr ← 0;  { we start with the string's first character }
  while (ex_buf_ptr < ex_buf_length) do
    begin if (ex_buf[ex_buf_ptr] = left_brace) then
      begin incr(brace_level);
      if (brace_level ≠ 1) then goto ok_pascal_i_give_up;
      if (ex_buf_ptr + 4 > ex_buf_length) then goto ok_pascal_i_give_up
      else if (ex_buf[ex_buf_ptr + 1] ≠ backslash) then goto ok_pascal_i_give_up;
      if (conversion_type = title_lowers) then
        if (ex_buf_ptr = 0) then goto ok_pascal_i_give_up
        else if ((prev_colon) ∧ (lex_class[ex_buf[ex_buf_ptr − 1]] = white_space)) then
            goto ok_pascal_i_give_up;
      ⟨ Convert a special character 371 ⟩;
    ok_pascal_i_give_up: prev_colon ← false;
      end
    else if (ex_buf[ex_buf_ptr] = right_brace) then
        begin decr_brace_level(pop_lit2); prev_colon ← false;
        end
      else if (brace_level = 0) then ⟨ Convert a brace_level = 0 character 376 ⟩;
    incr(ex_buf_ptr);
    end;
  check_brace_level(pop_lit2);
  end
```

This code is used in section 364.

371. We're dealing with a special character (usually either an undotted 'ı' or 'ȷ', or an accent like one in Table 3.1 of the LaTeX manual, or a foreign character like one in Table 3.2) if the first character after the *left_brace* is a *backslash*; the special character ends with the matching *right_brace*. How we handle what's in between depends on the special character. In general, this code will do reasonably well if there is other stuff, too, between braces, but it doesn't try to do anything special with *colon*s.

```
⟨ Convert a special character 371 ⟩ ≡
  begin incr(ex_buf_ptr); { skip over the left_brace }
  while ((ex_buf_ptr < ex_buf_length) ∧ (brace_level > 0)) do
    begin incr(ex_buf_ptr); { skip over the backslash }
    ex_buf_xptr ← ex_buf_ptr;
    while ((ex_buf_ptr < ex_buf_length) ∧ (lex_class[ex_buf[ex_buf_ptr]] = alpha)) do incr(ex_buf_ptr);
          { this scans the control sequence }
    control_seq_loc ← str_lookup(ex_buf, ex_buf_xptr, ex_buf_ptr − ex_buf_xptr, control_seq_ilk, dont_insert);
    if (hash_found) then ⟨ Convert the accented or foreign character, if necessary 372 ⟩;
    ex_buf_xptr ← ex_buf_ptr;
    while ((ex_buf_ptr < ex_buf_length) ∧ (brace_level > 0) ∧ (ex_buf[ex_buf_ptr] ≠ backslash)) do
      begin { this scans to the next control sequence }
      if (ex_buf[ex_buf_ptr] = right_brace) then decr(brace_level)
      else if (ex_buf[ex_buf_ptr] = left_brace) then incr(brace_level);
      incr(ex_buf_ptr);
      end;
    ⟨ Convert a noncontrol sequence 375 ⟩;
    end;
  decr(ex_buf_ptr); { unskip the right_brace }
  end
```

This code is used in section 370.

372. A control sequence, for the purposes of this program, consists just of the consecutive alphabetic characters following the *backslash*; it might be empty (although ones in this section aren't).

```
⟨ Convert the accented or foreign character, if necessary 372 ⟩ ≡
  begin case (conversion_type) of
  title_lowers, all_lowers: case (ilk_info[control_seq_loc]) of
    n_l_upper, n_o_upper, n_oe_upper, n_ae_upper, n_aa_upper:
          lower_case(ex_buf, ex_buf_xptr, ex_buf_ptr − ex_buf_xptr);
    othercases do_nothing
    endcases;
  all_uppers: case (ilk_info[control_seq_loc]) of
    n_l, n_o, n_oe, n_ae, n_aa: upper_case(ex_buf, ex_buf_xptr, ex_buf_ptr − ex_buf_xptr);
    n_i, n_j, n_ss: ⟨ Convert, then remove the control sequence 374 ⟩;
    othercases do_nothing
    endcases;
  bad_conversion: do_nothing;
  othercases case_conversion_confusion
  endcases;
  end
```

This code is used in section 371.

373. Another bug complaint.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
procedure case_conversion_confusion;
  begin confusion('Unknown␣type␣of␣case␣conversion');
  end;
```

374. After converting the control sequence, we need to remove the preceding *backslash* and any following *white_space*.

```
⟨ Convert, then remove the control sequence 374 ⟩ ≡
  begin upper_case(ex_buf, ex_buf_xptr, ex_buf_ptr − ex_buf_xptr);
  while (ex_buf_xptr < ex_buf_ptr) do
    begin { remove preceding backslash and shift down }
    ex_buf[ex_buf_xptr − 1] ← ex_buf[ex_buf_xptr]; incr(ex_buf_xptr);
    end;
  decr(ex_buf_xptr);
  while ((ex_buf_ptr < ex_buf_length) ∧ (lex_class[ex_buf[ex_buf_ptr]] = white_space)) do incr(ex_buf_ptr);
        { remove white_space trailing the control seq }
  tmp_ptr ← ex_buf_ptr;
  while (tmp_ptr < ex_buf_length) do
    begin { more shifting down }
    ex_buf[tmp_ptr − (ex_buf_ptr − ex_buf_xptr)] ← ex_buf[tmp_ptr]; incr(tmp_ptr)
    end;
  ex_buf_length ← tmp_ptr − (ex_buf_ptr − ex_buf_xptr); ex_buf_ptr ← ex_buf_xptr;
  end
```

This code is used in section 372.

375. There are no control sequences in what we're about to convert, so a straight conversion suffices.

```
⟨ Convert a noncontrol sequence 375 ⟩ ≡
  begin case (conversion_type) of
  title_lowers, all_lowers: lower_case(ex_buf, ex_buf_xptr, ex_buf_ptr − ex_buf_xptr);
  all_uppers: upper_case(ex_buf, ex_buf_xptr, ex_buf_ptr − ex_buf_xptr);
  bad_conversion: do_nothing;
  othercases case_conversion_confusion
  endcases;
  end
```

This code is used in section 371.

376. This code does any needed conversion for an ordinary character; it won't touch nonletters.

```
⟨ Convert a brace_level = 0 character 376 ⟩ ≡
  begin case (conversion_type) of
  title_lowers: begin if (ex_buf_ptr = 0) then do_nothing
    else if ((prev_colon) ∧ (lex_class[ex_buf[ex_buf_ptr − 1]] = white_space)) then do_nothing
      else lower_case(ex_buf, ex_buf_ptr, 1);
    if (ex_buf[ex_buf_ptr] = colon) then prev_colon ← true
    else if (lex_class[ex_buf[ex_buf_ptr]] ≠ white_space) then prev_colon ← false;
    end;
  all_lowers: lower_case(ex_buf, ex_buf_ptr, 1);
  all_uppers: upper_case(ex_buf, ex_buf_ptr, 1);
  bad_conversion: do_nothing;
  othercases case_conversion_confusion
  endcases;
  end
```

This code is used in section 370.

377. The *built_in* function `chr.to.int$` pops the top (string) literal, makes sure it's a single character, converts it to the corresponding *ASCII_code* integer, and pushes this integer. If the literal isn't an appropriate string, it complains and pushes the integer 0.

```
⟨ execute_fn(chr.to.int$) 377 ⟩ ≡
procedure x_chr_to_int;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(0, stk_int);
    end
  else if (length(pop_lit1) ≠ 1) then
      begin print('"'); print_pool_str(pop_lit1); bst_ex_warn('"␣isn''t␣a␣single␣character');
      push_lit_stk(0, stk_int);
      end
    else push_lit_stk(str_pool[str_start[pop_lit1]], stk_int);   { push the (ASCII_code) integer }
  end;
```

This code is used in section 342.

378. The *built_in* function `cite$` pushes the appropriate string from *cite_list* onto the stack.

```
⟨ execute_fn(cite$) 378 ⟩ ≡
procedure x_cite;
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else push_lit_stk(cur_cite_str, stk_str);
  end;
```

This code is used in section 342.

379. The *built_in* function `duplicate$` pops the top literal from the stack and pushes two copies of it.

```
⟨ execute_fn(duplicate$) 379 ⟩ ≡
procedure x_duplicate;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then
    begin push_lit_stk(pop_lit1, pop_typ1); push_lit_stk(pop_lit1, pop_typ1);
    end
  else begin repush_string;
    if (pop_lit1 < cmd_str_ptr) then push_lit_stk(pop_lit1, pop_typ1)
    else begin str_room(length(pop_lit1)); sp_ptr ← str_start[pop_lit1]; sp_end ← str_start[pop_lit1 + 1];
      while (sp_ptr < sp_end) do
        begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
        end;
      push_lit_stk(make_string, stk_str);   { and push it onto the stack }
      end;
    end;
  end;
```

This code is used in section 342.

380. The *built_in* function `empty$` pops the top literal and pushes the integer 1 if it's a missing field or a string having no non*white_space* characters, 0 otherwise. If the literal isn't a missing field or a string, it complains and pushes 0.

```
⟨ execute_fn(empty$) 380 ⟩ ≡
procedure x_empty;
  label exit;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  case (pop_typ1) of
  stk_str: ⟨ Push 0 if the string has a non white_space char, else 1 381 ⟩;
  stk_field_missing: push_lit_stk(1, stk_int);
  stk_empty: push_lit_stk(0, stk_int);
  othercases begin print_stk_lit(pop_lit1, pop_typ1);
    bst_ex_warn(´,␣not␣a␣string␣or␣missing␣field,´); push_lit_stk(0, stk_int);
    end
  endcases;
exit: end;
```

This code is used in section 342.

381. When we arrive here we're dealing with a legitimate string. If it has no characters, or has nothing but *white_space* characters, we push 1, otherwise we push 0.

```
⟨ Push 0 if the string has a non white_space char, else 1 381 ⟩ ≡
  begin sp_ptr ← str_start[pop_lit1]; sp_end ← str_start[pop_lit1 + 1];
  while (sp_ptr < sp_end) do
    begin if (lex_class[str_pool[sp_ptr]] ≠ white_space) then
      begin push_lit_stk(0, stk_int); return;
      end;
    incr(sp_ptr);
    end;
  push_lit_stk(1, stk_int);
  end
```

This code is used in section 380.

382. The *built_in* function `format.name$` pops the top three literals (they are a string, an integer, and a string literal, in that order). The last string literal represents a name list (each name corresponding to a person), the integer literal specifies which name to pick from this list, and the first string literal specifies how to format this name, as described in the BIBTEX documentation. Finally, this function pushes the formatted name. If any of the types is incorrect, it complains and pushes the null string.

```
  define von_found = 52  { for when a von token is found }
⟨ execute_fn(format.name$) 382 ⟩ ≡
procedure x_format_name;
  label loop1_exit, loop2_exit, von_found;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2); pop_lit_stk(pop_lit3, pop_typ3);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(s_null, stk_str);
    end
  else if (pop_typ2 ≠ stk_int) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int); push_lit_stk(s_null, stk_str);
      end
    else if (pop_typ3 ≠ stk_str) then
        begin print_wrong_stk_lit(pop_lit3, pop_typ3, stk_str); push_lit_stk(s_null, stk_str);
        end
      else begin ex_buf_length ← 0; add_buf_pool(pop_lit3); ⟨ Isolate the desired name 383 ⟩;
        ⟨ Copy name and count commas to determine syntax 387 ⟩;
        ⟨ Find the parts of the name 395 ⟩;
        ex_buf_length ← 0; add_buf_pool(pop_lit1); figure_out_the_formatted_name;
        add_pool_buf_and_push;  { push the formatted string onto the stack }
        end;
  end;
```

This code is used in section 342.

383. This module skips over undesired names in *pop_lit3* and it throws away the "and" from the end of the name if it exists. When it's done, *ex_buf_xptr* points to its first character and *ex_buf_ptr* points just past its last.

```
⟨ Isolate the desired name 383 ⟩ ≡
  begin ex_buf_ptr ← 0; num_names ← 0;
  while ((num_names < pop_lit2) ∧ (ex_buf_ptr < ex_buf_length)) do
    begin incr(num_names); ex_buf_xptr ← ex_buf_ptr; name_scan_for_and(pop_lit3);
    end;
  if (ex_buf_ptr < ex_buf_length) then  { remove the "and" }
    ex_buf_ptr ← ex_buf_ptr − 4;
  if (num_names < pop_lit2) then
    begin if (pop_lit2 = 1) then print('There␣is␣no␣name␣in␣"')
    else print('There␣aren''t␣', pop_lit2 : 0, '␣names␣in␣"');
    print_pool_str(pop_lit3); bst_ex_warn('"');
    end
  end
```

This code is used in section 382.

384. This module, starting at *ex_buf_ptr*, looks in *ex_buf* for an "and" surrounded by nonnull *white_space*. It stops either at *ex_buf_length* or just past the "and", whichever comes first, setting *ex_buf_ptr* accordingly. Its parameter *pop_lit_var* is either *pop_lit3* or *pop_lit1*, depending on whether `format.name$` or `num.names$` calls it.

```
⟨ Procedures and functions for name-string processing 367 ⟩ +≡
procedure name_scan_for_and(pop_lit_var : str_number);
  begin brace_level ← 0; preceding_white ← false; and_found ← false;
  while ((¬and_found) ∧ (ex_buf_ptr < ex_buf_length)) do
    case (ex_buf[ex_buf_ptr]) of
    "a", "A": begin incr(ex_buf_ptr);
      if (preceding_white) then ⟨ See if we have an "and" 386 ⟩;  { if so, and_found ← true }
      preceding_white ← false;
      end;
    left_brace: begin incr(brace_level); incr(ex_buf_ptr); ⟨ Skip over ex_buf stuff at brace_level > 0 385 ⟩;
      preceding_white ← false;
      end;
    right_brace: begin decr_brace_level(pop_lit_var);  { this checks for an error }
      incr(ex_buf_ptr); preceding_white ← false;
      end;
    othercases if (lex_class[ex_buf[ex_buf_ptr]] = white_space) then
        begin incr(ex_buf_ptr); preceding_white ← true;
        end
      else begin incr(ex_buf_ptr); preceding_white ← false;
        end
    endcases;
  check_brace_level(pop_lit_var);
  end;
```

385. When we come here *ex_buf_ptr* is just past the *left_brace*, and when we leave it's either at *ex_buf_length* or just past the matching *right_brace*.

```
⟨ Skip over ex_buf stuff at brace_level > 0 385 ⟩ ≡
  while ((brace_level > 0) ∧ (ex_buf_ptr < ex_buf_length)) do
    begin if (ex_buf[ex_buf_ptr] = right_brace) then decr(brace_level)
    else if (ex_buf[ex_buf_ptr] = left_brace) then incr(brace_level);
    incr(ex_buf_ptr);
    end
```

This code is used in section 384.

386. When we come here *ex_buf_ptr* is just past the "a" or "A", and when we leave it's either at the same place or, if we found an "and", at the following *white_space* character.

```
⟨ See if we have an "and" 386 ⟩ ≡
  begin if (ex_buf_ptr ≤ (ex_buf_length − 3)) then  { enough characters are left }
    if ((ex_buf[ex_buf_ptr] = "n") ∨ (ex_buf[ex_buf_ptr] = "N")) then
      if ((ex_buf[ex_buf_ptr + 1] = "d") ∨ (ex_buf[ex_buf_ptr + 1] = "D")) then
        if (lex_class[ex_buf[ex_buf_ptr + 2]] = white_space) then
          begin ex_buf_ptr ← ex_buf_ptr + 2; and_found ← true;
          end;
  end
```

This code is used in section 384.

387. When we arrive here, the desired name is in *ex_buf*[*ex_buf_xptr*] through *ex_buf*[*ex_buf_ptr* − 1]. This module does its thing for characters only at *brace_level* = 0; the rest get processed verbatim. It removes leading *white_space* (and *sep_char*s), and trailing *white_space* (and *sep_char*s) and *comma*s, complaining for each trailing *comma*. It then copies the name into *name_buf*, removing all *white_space*, *sep_char*s and *comma*s, counting *comma*s, and constructing a list of name tokens, which are sequences of characters separated (at *brace_level* = 0) by *white_space*, *sep_char*s or *comma*s. Each name token but the first has an associated *name_sep_char*, the character that separates it from the preceding token. If there are too many (more than two) *comma*s, a complaint is in order.

```
⟨ Copy name and count comma's to determine syntax 387 ⟩ ≡
  begin ⟨ Remove leading and trailing junk, complaining if necessary 388 ⟩;
  name_bf_ptr ← 0; num_commas ← 0; num_tokens ← 0;
  token_starting ← true; { to indicate that a name token is starting }
  while (ex_buf_xptr < ex_buf_ptr) do
    case (ex_buf[ex_buf_xptr]) of
    comma: ⟨ Name-process a comma 389 ⟩;
    left_brace: ⟨ Name-process a left_brace 390 ⟩;
    right_brace: ⟨ Name-process a right_brace 391 ⟩;
    othercases case (lex_class[ex_buf[ex_buf_xptr]]) of
      white_space: ⟨ Name-process a white_space 392 ⟩;
      sep_char: ⟨ Name-process a sep_char 393 ⟩;
      othercases ⟨ Name-process some other character 394 ⟩
      endcases
    endcases;
  name_tok[num_tokens] ← name_bf_ptr; { this is an end-marker }
  end
```

This code is used in section 382.

388. This module removes all leading *white_space* (and *sep_char*s), and trailing *white_space* (and *sep_char*s) and *comma*s. It complains for each trailing *comma*.

```
⟨ Remove leading and trailing junk, complaining if necessary 388 ⟩ ≡
  begin while ((ex_buf_xptr < ex_buf_ptr) ∧ (lex_class[ex_buf[ex_buf_ptr]] =
        white_space) ∧ (lex_class[ex_buf[ex_buf_ptr]] = sep_char)) do incr(ex_buf_xptr);
        { this removes leading stuff }
  while (ex_buf_ptr > ex_buf_xptr) do { now remove trailing stuff }
    case (lex_class[ex_buf[ex_buf_ptr − 1]]) of
    white_space, sep_char: decr(ex_buf_ptr);
    othercases if (ex_buf[ex_buf_ptr − 1] = comma) then
        begin print('Name␣', pop_lit2 : 0, '␣in␣"'); print_pool_str(pop_lit3);
        print('"␣has␣a␣comma␣at␣the␣end'); bst_ex_warn_print; decr(ex_buf_ptr);
        end
      else goto loop1_exit
    endcases;
loop1_exit: end
```

This code is used in section 387.

389. Here we mark the token number at which this comma has occurred.

```
⟨ Name-process a comma 389 ⟩ ≡
  begin if (num_commas = 2) then
    begin print(´Too␣many␣commas␣in␣name␣´, pop_lit2 : 0, ´␣of␣"´); print_pool_str(pop_lit3);
    print(´"´); bst_ex_warn_print;
    end
  else begin incr(num_commas);
    if (num_commas = 1) then comma1 ← num_tokens
    else comma2 ← num_tokens; { num_commas = 2 }
    name_sep_char[num_tokens] ← comma;
    end;
  incr(ex_buf_xptr); token_starting ← true;
  end
```

This code is used in section 387.

390. We copy the stuff up through the matching *right_brace* verbatim.

```
⟨ Name-process a left_brace 390 ⟩ ≡
  begin incr(brace_level);
  if (token_starting) then
    begin name_tok[num_tokens] ← name_bf_ptr; incr(num_tokens);
    end;
  name_buf[name_bf_ptr] ← ex_buf[ex_buf_xptr]; incr(name_bf_ptr); incr(ex_buf_xptr);
  while ((brace_level > 0) ∧ (ex_buf_xptr < ex_buf_ptr)) do
    begin if (ex_buf[ex_buf_xptr] = right_brace) then decr(brace_level)
    else if (ex_buf[ex_buf_xptr] = left_brace) then incr(brace_level);
    name_buf[name_bf_ptr] ← ex_buf[ex_buf_xptr]; incr(name_bf_ptr); incr(ex_buf_xptr);
    end;
  token_starting ← false;
  end
```

This code is used in section 387.

391. We don't copy an extra *right_brace*; this code will almost never be executed.

```
⟨ Name-process a right_brace 391 ⟩ ≡
  begin if (token_starting) then
    begin name_tok[num_tokens] ← name_bf_ptr; incr(num_tokens);
    end;
  print(´Name␣´, pop_lit2 : 0, ´␣of␣"´); print_pool_str(pop_lit3);
  bst_ex_warn(´"␣isn´´t␣brace␣balanced´); incr(ex_buf_xptr); token_starting ← false;
  end
```

This code is used in section 387.

392. A token will be starting soon in a buffer near you, one way...

```
⟨ Name-process a white_space 392 ⟩ ≡
  begin if (¬token_starting) then name_sep_char[num_tokens] ← space;
  incr(ex_buf_xptr); token_starting ← true;
  end
```

This code is used in section 387.

393. or another. If one of the valid *sep_char*s appears between tokens, we usually use it instead of a *space*. If the user has been silly enough to have multiple *sep_char*s, or to have both *white_space* and a *sep_char*, we use the first such character.

```
⟨ Name-process a sep_char 393 ⟩ ≡
  begin if (¬token_starting) then name_sep_char[num_tokens] ← ex_buf[ex_buf_xptr];
  incr(ex_buf_xptr); token_starting ← true;
  end
```

This code is used in section 387.

394. For ordinary characters, we just copy the character.

```
⟨ Name-process some other character 394 ⟩ ≡
  begin if (token_starting) then
    begin name_tok[num_tokens] ← name_bf_ptr; incr(num_tokens);
    end;
  name_buf[name_bf_ptr] ← ex_buf[ex_buf_xptr]; incr(name_bf_ptr); incr(ex_buf_xptr);
  token_starting ← false;
  end
```

This code is used in section 387.

395. Here we set all the pointers for the various parts of the name, depending on which of the three possible syntaxes this name uses.

```
⟨ Find the parts of the name 395 ⟩ ≡
  begin if (num_commas = 0) then
    begin first_start ← 0; last_end ← num_tokens; jr_end ← last_end;
    ⟨ Determine where the first name ends and von name starts and ends 396 ⟩;
    end
  else if (num_commas = 1) then
      begin von_start ← 0; last_end ← comma1; jr_end ← last_end; first_start ← jr_end;
      first_end ← num_tokens; von_name_ends_and_last_name_starts_stuff;
      end
    else if (num_commas = 2) then
        begin von_start ← 0; last_end ← comma1; jr_end ← comma2; first_start ← jr_end;
        first_end ← num_tokens; von_name_ends_and_last_name_starts_stuff;
        end
      else confusion('Illegal␣number␣of␣comma,s');
  end
```

This code is used in section 382.

396. When there are no brace-level-0 *comma*s in the name, the von name starts with the first nonlast token whose first brace-level-0 letter is in lower case (for the purposes of this determination, an accented or foreign character at brace-level-1 that's in lower case will do, as well). A module following this one determines where the von name ends and the last starts.

```
⟨ Determine where the first name ends and von name starts and ends 396 ⟩ ≡
  begin von_start ← 0;
  while (von_start < last_end − 1) do
    begin name_bf_ptr ← name_tok[von_start]; name_bf_xptr ← name_tok[von_start + 1];
    if (von_token_found) then
      begin von_name_ends_and_last_name_starts_stuff; goto von_found;
      end;
    incr(von_start);
    end; { there's no von name, so }
  while (von_start > 0) do { backtrack if there are connected tokens }
    begin if ((lex_class[name_sep_char[von_start]] ≠ sep_char) ∨ (name_sep_char[von_start] = tie)) then
      goto loop2_exit;
    decr(von_start);
    end;
loop2_exit: von_end ← von_start;
von_found: first_end ← von_start;
  end
```

This code is used in section 395.

397. It's a von token if there exists a first brace-level-0 letter (or brace-level-1 special character), and it's in lower case; in this case we return *true*. The token is in *name_buf*, starting at *name_bf_ptr* and ending just before *name_bf_xptr*.

```
  define return_von_found ≡
           begin von_token_found ← true; return;
           end

⟨ Procedures and functions for name-string processing 367 ⟩ +≡
function von_token_found: boolean;
  label exit;
  begin nm_brace_level ← 0; von_token_found ← false; { now it's easy to exit if necessary }
  while (name_bf_ptr < name_bf_xptr) do
    if ((name_buf[name_bf_ptr] ≥ "A") ∧ (name_buf[name_bf_ptr] ≤ "Z")) then return
    else if ((name_buf[name_bf_ptr] ≥ "a") ∧ (name_buf[name_bf_ptr] ≤ "z")) then return_von_found
      else if (name_buf[name_bf_ptr] = left_brace) then
           begin incr(nm_brace_level); incr(name_bf_ptr);
           if ((name_bf_ptr + 2 < name_bf_xptr) ∧ (name_buf[name_bf_ptr] = backslash)) then
             ⟨ Check the special character (and return) 398 ⟩
           else ⟨ Skip over name_buf stuff at nm_brace_level > 0 400 ⟩;
           end
        else incr(name_bf_ptr);
exit: end;
```

398. When we come here *name_bf_ptr* is just past the *left_brace*, but we always leave by **return**ing.

⟨ Check the special character (and **return**) 398 ⟩ ≡
```
begin incr(name_bf_ptr);  { skip over the backslash }
name_bf_yptr ← name_bf_ptr;
while ((name_bf_ptr < name_bf_xptr) ∧ (lex_class[name_buf[name_bf_ptr]] = alpha)) do
  incr(name_bf_ptr);  { this scans the control sequence }
control_seq_loc ← str_lookup(name_buf, name_bf_yptr, name_bf_ptr − name_bf_yptr, control_seq_ilk,
    dont_insert);
if (hash_found) then ⟨ Handle this accented or foreign character (and return) 399 ⟩;
while ((name_bf_ptr < name_bf_xptr) ∧ (nm_brace_level > 0)) do
  begin if ((name_buf[name_bf_ptr] ≥ "A") ∧ (name_buf[name_bf_ptr] ≤ "Z")) then return
  else if ((name_buf[name_bf_ptr] ≥ "a") ∧ (name_buf[name_bf_ptr] ≤ "z")) then return_von_found
    else if (name_buf[name_bf_ptr] = right_brace) then decr(nm_brace_level)
      else if (name_buf[name_bf_ptr] = left_brace) then incr(nm_brace_level);
  incr(name_bf_ptr);
  end;
return;
end
```
This code is used in section 397.

399. The accented or foreign character is either '`\i`' or '`\j`' or one of the eleven alphabetic foreign characters in Table 3.2 of the LaTeX manual.

⟨ Handle this accented or foreign character (and **return**) 399 ⟩ ≡
```
begin case (ilk_info[control_seq_loc]) of
n_oe_upper, n_ae_upper, n_aa_upper, n_o_upper, n_l_upper: return;
n_i, n_j, n_oe, n_ae, n_aa, n_o, n_l, n_ss: return_von_found;
othercases confusion(´Control-sequence␣hash␣error´)
endcases;
end
```
This code is used in section 398.

400. When we come here *name_bf_ptr* is just past the *left_brace*; when we leave it's either at *name_bf_xptr* or just past the matching *right_brace*.

⟨ Skip over *name_buf* stuff at *nm_brace_level* > 0 400 ⟩ ≡
```
while ((nm_brace_level > 0) ∧ (name_bf_ptr < name_bf_xptr)) do
  begin if (name_buf[name_bf_ptr] = right_brace) then decr(nm_brace_level)
  else if (name_buf[name_bf_ptr] = left_brace) then incr(nm_brace_level);
  incr(name_bf_ptr);
  end
```
This code is used in section 397.

401. The last name starts just past the last token, before the first *comma* (if there is no *comma*, there is deemed to be one at the end of the string), for which there exists a first brace-level-0 letter (or brace-level-1 special character), and it's in lower case, unless this last token is also the last token before the *comma*, in which case the last name starts with this token (unless this last token is connected by a *sep_char* other than a *tie* to the previous token, in which case the last name starts with as many tokens earlier as are connected by non*tie*s to this last one (except on Tuesdays ...), although this module never sees such a case). Note that if there are any tokens in either the von or last names, then the last name has at least one, even if it starts with a lower-case letter.

```
⟨ Procedures and functions for name-string processing 367 ⟩ +≡
procedure von_name_ends_and_last_name_starts_stuff;
  label exit;
  begin { there may or may not be a von name }
  von_end ← last_end − 1;
  while (von_end > von_start) do
    begin name_bf_ptr ← name_tok[von_end − 1]; name_bf_xptr ← name_tok[von_end];
    if (von_token_found) then return;
    decr(von_end);
    end;
exit: end;
```

402. This module uses the information in *pop_lit1* to format the name. Everything at *sp_brace_level* = 0 is copied verbatim to the formatted string; the rest is described in the succeeding modules.

```
⟨ Figure out the formatted name 402 ⟩ ≡
  begin ex_buf_ptr ← 0; sp_brace_level ← 0; sp_ptr ← str_start[pop_lit1]; sp_end ← str_start[pop_lit1 + 1];
  while (sp_ptr < sp_end) do
    if (str_pool[sp_ptr] = left_brace) then
      begin incr(sp_brace_level); incr(sp_ptr); ⟨ Format this part of the name 403 ⟩;
      end
    else if (str_pool[sp_ptr] = right_brace) then
        begin braces_unbalanced_complaint(pop_lit1); incr(sp_ptr);
        end
      else begin append_ex_buf_char_and_check(str_pool[sp_ptr]); incr(sp_ptr);
        end;
  if (sp_brace_level > 0) then braces_unbalanced_complaint(pop_lit1);
  ex_buf_length ← ex_buf_ptr;
  end
```

This code is used in section 420.

403. When we arrive here we're at *sp_brace_level* = 1, just past the *left_brace*. Letters at this *sp_brace_level* other than those denoting the parts of the name (i.e., the first letters of 'first,' 'last,' 'von,' and 'jr,' ignoring case) are illegal. We do two passes over this group; the first determines whether we're to output anything, and, if we are, the second actually outputs it.

```
⟨ Format this part of the name 403 ⟩ ≡
  begin sp_xptr1 ← sp_ptr; alpha_found ← false; double_letter ← false; end_of_group ← false;
  to_be_written ← true;
  while ((¬end_of_group) ∧ (sp_ptr < sp_end)) do
    if (lex_class[str_pool[sp_ptr]] = alpha) then
      begin incr(sp_ptr); ⟨ Figure out what this letter means 405 ⟩;
      end
    else if (str_pool[sp_ptr] = right_brace) then
        begin decr(sp_brace_level); incr(sp_ptr); end_of_group ← true;
        end
      else if (str_pool[sp_ptr] = left_brace) then
          begin incr(sp_brace_level); incr(sp_ptr); skip_stuff_at_sp_brace_level_greater_than_one;
          end
        else incr(sp_ptr);
  if ((end_of_group) ∧ (to_be_written)) then   { do the second pass }
    ⟨ Finally format this part of the name 411 ⟩;
  end
```

This code is used in section 402.

404. When we come here *sp_ptr* is just past the *left_brace*, and when we leave it's either at *sp_end* or just past the matching *right_brace*.

```
⟨ Procedures and functions for name-string processing 367 ⟩ +≡
procedure skip_stuff_at_sp_brace_level_greater_than_one;
  begin while ((sp_brace_level > 1) ∧ (sp_ptr < sp_end)) do
    begin if (str_pool[sp_ptr] = right_brace) then decr(sp_brace_level)
    else if (str_pool[sp_ptr] = left_brace) then incr(sp_brace_level);
    incr(sp_ptr);
    end;
  end;
```

405. We won't output anything for this part of the name if this is a second occurrence of an *sp_brace_level* = 1 letter, if it's an illegal letter, or if there are no tokens corresponding to this part. We also determine if we're we to output complete tokens (indicated by a double letter).

```
⟨ Figure out what this letter means 405 ⟩ ≡
  begin if (alpha_found) then
    begin brace_lvl_one_letters_complaint; to_be_written ← false;
    end
  else begin case (str_pool[sp_ptr − 1]) of
    "f","F": ⟨ Figure out what tokens we'll output for the 'first' name 407 ⟩;
    "v","V": ⟨ Figure out what tokens we'll output for the 'von' name 408 ⟩;
    "l","L": ⟨ Figure out what tokens we'll output for the 'last' name 409 ⟩;
    "j","J": ⟨ Figure out what tokens we'll output for the 'jr' name 410 ⟩;
    othercases begin brace_lvl_one_letters_complaint; to_be_written ← false;
      end
    endcases;
    if (double_letter) then incr(sp_ptr);
    end;
  alpha_found ← true;
  end
```

This code is used in section 403.

406. At most one of the important letters, perhaps doubled, may appear at *sp_brace_level* = 1.

```
⟨ Procedures and functions for name-string processing 367 ⟩ +≡
procedure brace_lvl_one_letters_complaint;
  begin print(´The␣format␣string␣"´); print_pool_str(pop_lit1);
  bst_ex_warn(´"␣has␣an␣illegal␣brace-level-1␣letter´);
  end;
```

407. Here we set pointers into *name_tok* and note whether we'll be dealing with a full first-name tokens (*double_letter* = *true*) or abbreviations (*double_letter* = *false*).

```
⟨ Figure out what tokens we'll output for the 'first' name 407 ⟩ ≡
  begin cur_token ← first_start; last_token ← first_end;
  if (cur_token = last_token) then to_be_written ← false;
  if ((str_pool[sp_ptr] = "f") ∨ (str_pool[sp_ptr] = "F")) then double_letter ← true;
  end
```

This code is used in section 405.

408. The same as above but for von-name tokens.

```
⟨ Figure out what tokens we'll output for the 'von' name 408 ⟩ ≡
  begin cur_token ← von_start; last_token ← von_end;
  if (cur_token = last_token) then to_be_written ← false;
  if ((str_pool[sp_ptr] = "v") ∨ (str_pool[sp_ptr] = "V")) then double_letter ← true;
  end
```

This code is used in section 405.

409. The same as above but for last-name tokens.

```
⟨ Figure out what tokens we'll output for the 'last' name 409 ⟩ ≡
  begin cur_token ← von_end; last_token ← last_end;
  if (cur_token = last_token) then to_be_written ← false;
  if ((str_pool[sp_ptr] = "l") ∨ (str_pool[sp_ptr] = "L")) then double_letter ← true;
  end
```

This code is used in section 405.

410. The same as above but for jr-name tokens.

```
⟨ Figure out what tokens we'll output for the 'jr' name 410 ⟩ ≡
  begin cur_token ← last_end; last_token ← jr_end;
  if (cur_token = last_token) then to_be_written ← false;
  if ((str_pool[sp_ptr] = "j") ∨ (str_pool[sp_ptr] = "J")) then double_letter ← true;
  end
```

This code is used in section 405.

411. This is the second pass over this part of the name; here we actually write stuff out to *ex_buf*.

```
⟨ Finally format this part of the name 411 ⟩ ≡
  begin ex_buf_xptr ← ex_buf_ptr; sp_ptr ← sp_xptr1; sp_brace_level ← 1;
  while (sp_brace_level > 0) do
    if ((lex_class[str_pool[sp_ptr]] = alpha) ∧ (sp_brace_level = 1)) then
      begin incr(sp_ptr); ⟨ Figure out how to output the name tokens, and do it 412 ⟩;
      end
    else if (str_pool[sp_ptr] = right_brace) then
        begin decr(sp_brace_level); incr(sp_ptr);
        if (sp_brace_level > 0) then append_ex_buf_char_and_check(right_brace);
        end
      else if (str_pool[sp_ptr] = left_brace) then
          begin incr(sp_brace_level); incr(sp_ptr); append_ex_buf_char_and_check(left_brace);
          end
        else begin append_ex_buf_char_and_check(str_pool[sp_ptr]); incr(sp_ptr);
          end;
  if (ex_buf_ptr > 0) then
    if (ex_buf[ex_buf_ptr − 1] = tie) then ⟨ Handle a discretionary tie 419 ⟩;
  end
```

This code is used in section 403.

412. When we come here, *sp_ptr* is just past the letter indicating the part of the name for which we're about to output tokens. When we leave, it's at the first character of the rest of the group.

```
⟨ Figure out how to output the name tokens, and do it 412 ⟩ ≡
  begin if (double_letter) then incr(sp_ptr);
  use_default ← true; sp_xptr2 ← sp_ptr;
  if (str_pool[sp_ptr] = left_brace) then { find the inter-token string }
    begin use_default ← false; incr(sp_brace_level); incr(sp_ptr); sp_xptr1 ← sp_ptr;
    skip_stuff_at_sp_brace_level_greater_than_one; sp_xptr2 ← sp_ptr − 1;
    end;
  ⟨ Finally output the name tokens 413 ⟩;
  if (¬use_default) then sp_ptr ← sp_xptr2 + 1;
  end
```

This code is used in section 411.

413. Here, for each token in this part, we output either a full or an abbreviated token and the inter-token string for all but the last token of this part.

```
⟨ Finally output the name tokens 413 ⟩ ≡
  while (cur_token < last_token) do
    begin if (double_letter) then ⟨ Finally output a full token 414 ⟩
    else ⟨ Finally output an abbreviated token 415 ⟩;
    incr(cur_token);
    if (cur_token < last_token) then ⟨ Finally output the inter-token string 417 ⟩;
    end
```

This code is used in section 412.

414. Here we output all the characters in the token, verbatim.

```
⟨ Finally output a full token 414 ⟩ ≡
  begin name_bf_ptr ← name_tok[cur_token]; name_bf_xptr ← name_tok[cur_token + 1];
  if (ex_buf_length + (name_bf_xptr − name_bf_ptr) > buf_size) then buffer_overflow;
  while (name_bf_ptr < name_bf_xptr) do
    begin append_ex_buf_char(name_buf[name_bf_ptr]); incr(name_bf_ptr);
    end;
  end
```

This code is used in section 413.

415. Here we output the first alphabetic or special character of the token; brace level is irrelevant for an alphabetic (but not a special) character.

```
⟨ Finally output an abbreviated token 415 ⟩ ≡
  begin name_bf_ptr ← name_tok[cur_token]; name_bf_xptr ← name_tok[cur_token + 1];
  while (name_bf_ptr < name_bf_xptr) do
    begin if (lex_class[name_buf[name_bf_ptr]] = alpha) then
      begin append_ex_buf_char_and_check(name_buf[name_bf_ptr]); goto loop_exit;
      end
    else if ((name_buf[name_bf_ptr] = left_brace) ∧ (name_bf_ptr + 1 < name_bf_xptr)) then
        if (name_buf[name_bf_ptr + 1] = backslash) then
          ⟨ Finally output a special character and exit loop 416 ⟩;
    incr(name_bf_ptr);
    end;
loop_exit: end
```

This code is used in section 413.

416. We output a special character here even if the user has been silly enough to make it nonalphabetic (and even if the user has been sillier still by not having a matching *right_brace*).

```
⟨ Finally output a special character and exit loop 416 ⟩ ≡
  begin if (ex_buf_ptr + 2 > buf_size) then buffer_overflow;
  append_ex_buf_char(left_brace); append_ex_buf_char(backslash); name_bf_ptr ← name_bf_ptr + 2;
  nm_brace_level ← 1;
  while ((name_bf_ptr < name_bf_xptr) ∧ (nm_brace_level > 0)) do
    begin if (name_buf[name_bf_ptr] = right_brace) then decr(nm_brace_level)
    else if (name_buf[name_bf_ptr] = left_brace) then incr(nm_brace_level);
    append_ex_buf_char_and_check(name_buf[name_bf_ptr]); incr(name_bf_ptr);
    end;
  goto loop_exit;
  end
```

This code is used in section 415.

417. Here we output either the `.bst` given string if it exists, or else the `.bib` *sep_char* if it exists, or else the default string. A *tie* is the default space character between the last two tokens of the name part, and between the first two tokens if the first token is short enough; otherwise, a *space* is the default.

```
define long_token = 3  { a token this length or longer is "long" }
⟨ Finally output the inter-token string 417 ⟩ ≡
  begin if (use_default) then
    begin if (¬double_letter) then append_ex_buf_char_and_check(period);
    if (lex_class[name_sep_char[cur_token]] = sep_char) then
      append_ex_buf_char_and_check(name_sep_char[cur_token])
    else if ((cur_token = last_token − 1) ∨ (¬enough_text_chars(long_token))) then
        append_ex_buf_char_and_check(tie)
      else append_ex_buf_char_and_check(space);
    end
  else begin if (ex_buf_length + (sp_xptr2 − sp_xptr1) > buf_size) then buffer_overflow;
    sp_ptr ← sp_xptr1;
    while (sp_ptr < sp_xptr2) do
      begin append_ex_buf_char(str_pool[sp_ptr]); incr(sp_ptr);
      end
    end;
  end
```

This code is used in section 413.

418. This function looks at the string in *ex_buf*, starting at *ex_buf_xptr* and ending just before *ex_buf_ptr*, and it returns *true* if there are *enough_chars*, where a special character (even if it's missing its matching *right_brace*) counts as a single charcter. This procedure is called only for strings that don't have too many *right_brace*s.

```
⟨ Procedures and functions for name-string processing 367 ⟩ +≡
function enough_text_chars(enough_chars : buf_pointer): boolean;
  begin num_text_chars ← 0; ex_buf_yptr ← ex_buf_xptr;
  while ((ex_buf_yptr < ex_buf_ptr) ∧ (num_text_chars < enough_chars)) do
    begin incr(ex_buf_yptr);
    if (ex_buf[ex_buf_yptr − 1] = left_brace) then
      begin incr(brace_level);
      if ((brace_level = 1) ∧ (ex_buf_yptr < ex_buf_ptr)) then
        if (ex_buf[ex_buf_yptr] = backslash) then
          begin incr(ex_buf_yptr);  { skip over the backslash }
          while ((ex_buf_yptr < ex_buf_ptr) ∧ (brace_level > 0)) do
            begin if (ex_buf[ex_buf_yptr] = right_brace) then decr(brace_level)
            else if (ex_buf[ex_buf_yptr] = left_brace) then incr(brace_level);
            incr(ex_buf_yptr);
            end;
          end;
      end
    else if (ex_buf[ex_buf_yptr − 1] = right_brace) then decr(brace_level);
    incr(num_text_chars);
    end;
  if (num_text_chars < enough_chars) then enough_text_chars ← false
  else enough_text_chars ← true;
  end;
```

419. If the last character output for this name part is a *tie* but the previous character it isn't, we're dealing with a discretionary *tie*; thus we replace it by a *space* if there are enough characters in the rest of the name part.

```
define long_name = 3  { a name this length or longer is "long" }
⟨ Handle a discretionary tie 419 ⟩ ≡
  begin decr(ex_buf_ptr);  { remove the previous tie }
  if (ex_buf[ex_buf_ptr − 1] = tie) then  { it's not a discretionary tie }
    do_nothing
  else if (¬enough_text_chars(long_name)) then  { this is a short name part }
      incr(ex_buf_ptr)  { so restore the tie }
    else  { replace it by a space }
  append_ex_buf_char(space);
  end
```

This code is used in section 411.

420. This is a procedure so that *x_format_name* is smaller.

```
⟨ Procedures and functions for name-string processing 367 ⟩ +≡
procedure figure_out_the_formatted_name;
  label loop_exit;
  begin ⟨ Figure out the formatted name 402 ⟩;
  end;
```

421. The *built_in* function `if$` pops the top three literals (they are two function literals and an integer literal, in that order); if the integer is greater than 0, it executes the second literal, else it executes the first. If any of the types is incorrect, it complains but does nothing else.

```
⟨ execute_fn(if$) 421 ⟩ ≡
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2); pop_lit_stk(pop_lit3, pop_typ3);
  if (pop_typ1 ≠ stk_fn) then print_wrong_stk_lit(pop_lit1, pop_typ1, stk_fn)
  else if (pop_typ2 ≠ stk_fn) then print_wrong_stk_lit(pop_lit2, pop_typ2, stk_fn)
    else if (pop_typ3 ≠ stk_int) then print_wrong_stk_lit(pop_lit3, pop_typ3, stk_int)
      else if (pop_lit3 > 0) then execute_fn(pop_lit2)
        else execute_fn(pop_lit1);
  end
```

This code is used in section 341.

422. The *built_in* function `int.to.chr$` pops the top (integer) literal, interpreted as the *ASCII_code* of a single character, converts it to the corresponding single-character string, and pushes this string. If the literal isn't an appropriate integer, it complains and pushes the null string.

```
⟨ execute_fn(int.to.chr$) 422 ⟩ ≡
procedure x_int_to_chr;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(s_null, stk_str);
    end
  else if ((pop_lit1 < 0) ∨ (pop_lit1 > 127)) then
      begin bst_ex_warn(pop_lit1 : 0, ´␣isn´´t␣valid␣ASCII´); push_lit_stk(s_null, stk_str);
      end
    else begin str_room(1); append_char(pop_lit1); push_lit_stk(make_string, stk_str);
      end;
  end;
```

This code is used in section 342.

423. The *built_in* function `int.to.str$` pops the top (integer) literal, converts it to its (unique) string equivalent, and pushes this string. If the literal isn't an integer, it complains and pushes the null string.

```
⟨ execute_fn(int.to.str$) 423 ⟩ ≡
procedure x_int_to_str;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(s_null, stk_str);
    end
  else begin int_to_ASCII(pop_lit1, ex_buf, 0, ex_buf_length);
    add_pool_buf_and_push; { push this string onto the stack }
    end;
  end;
```

This code is used in section 342.

424. The *built_in* function `missing$` pops the top literal and pushes the integer 1 if it's a missing field, 0 otherwise. If the literal isn't a missing field or a string, it complains and pushes 0. Unlike `empty$`, this function should be called only when *mess_with_entries* is true.

```
⟨ execute_fn(missing$) 424 ⟩ ≡
procedure x_missing;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else if ((pop_typ1 ≠ stk_str) ∧ (pop_typ1 ≠ stk_field_missing)) then
      begin if (pop_typ1 ≠ stk_empty) then
        begin print_stk_lit(pop_lit1, pop_typ1); bst_ex_warn('␣not␣a␣string␣or␣missing␣field,');
        end;
      push_lit_stk(0, stk_int);
      end
    else if (pop_typ1 = stk_field_missing) then push_lit_stk(1, stk_int)
      else push_lit_stk(0, stk_int);
  end;
```

This code is used in section 342.

425. The *built_in* function `newline$` writes whatever has accumulated in the output buffer *out_buf* onto the `.bbl` file.

```
⟨ execute_fn(newline$) 425 ⟩ ≡
  begin output_bbl_line;
  end
```

This code is used in section 341.

426. The *built_in* function `num.names$` pops the top (string) literal; it pushes the number of names the string represents—one plus the number of occurrences of the substring "and" (ignoring case differences) surrounded by nonnull *white_space* at the top brace level. If the literal isn't a string, it complains and pushes the value 0.

```
⟨ execute_fn(num.names$) 426 ⟩ ≡
procedure x_num_names;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(0, stk_int);
    end
  else begin ex_buf_length ← 0; add_buf_pool(pop_lit1); ⟨ Determine the number of names 427 ⟩;
    push_lit_stk(num_names, stk_int);
    end;
  end;
```

This code is used in section 342.

427. This module, while scanning the list of names, counts the occurrences of "and" (ignoring case differences) surrounded by nonnull *white_space*, and adds 1.

```
⟨ Determine the number of names 427 ⟩ ≡
  begin ex_buf_ptr ← 0; num_names ← 0;
  while (ex_buf_ptr < ex_buf_length) do
    begin name_scan_for_and(pop_lit1); incr(num_names);
    end;
  end
```

This code is used in section 426.

428. The *built_in* function `pop$` pops the top of the stack but doesn't print it.

```
⟨ execute_fn(pop$) 428 ⟩ ≡
  begin pop_lit_stk(pop_lit1, pop_typ1);
  end
```

This code is used in section 341.

429. The *built_in* function `preamble$` pushes onto the stack the concatenation of all the `preamble` strings read from the database files.

```
⟨ execute_fn(preamble$) 429 ⟩ ≡
procedure x_preamble;
  begin ex_buf_length ← 0; preamble_ptr ← 0;
  while (preamble_ptr < num_preamble_strings) do
    begin add_buf_pool(s_preamble[preamble_ptr]); incr(preamble_ptr);
    end;
  add_pool_buf_and_push; { push the concatenation string onto the stack }
  end;
```

This code is used in section 342.

430. The *built_in* function `purify$` pops the top (string) literal, removes nonalphanumeric characters except for *white_space* and *sep_char* characters (these get converted to a *space*) and removes certain alphabetic characters contained in the control sequences associated with a special character, and pushes the resulting string. If the literal isn't a string, it complains and pushes the null string.

```
⟨ execute_fn(purify$) 430 ⟩ ≡
procedure x_purify;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(s_null, stk_str);
    end
  else begin ex_buf_length ← 0; add_buf_pool(pop_lit1); ⟨ Perform the purification 431 ⟩;
    add_pool_buf_and_push; { push this string onto the stack }
    end;
  end;
```

This code is used in section 342.

431. The resulting string has nonalphanumeric characters removed, and each *white_space* or *sep_char* character converted to a *space*. The next module handles special characters. This code doesn't complain if the string isn't brace balanced.

```
⟨ Perform the purification 431 ⟩ ≡
  begin brace_level ← 0; { this is the top level }
  ex_buf_xptr ← 0; { this pointer is for the purified string }
  ex_buf_ptr ← 0; { and this one is for the original string }
  while (ex_buf_ptr < ex_buf_length) do
    begin case (lex_class[ex_buf[ex_buf_ptr]]) of
    white_space, sep_char: begin ex_buf[ex_buf_xptr] ← space; incr(ex_buf_xptr);
      end;
    alpha, numeric: begin ex_buf[ex_buf_xptr] ← ex_buf[ex_buf_ptr]; incr(ex_buf_xptr);
      end;
    othercases if (ex_buf[ex_buf_ptr] = left_brace) then
        begin incr(brace_level);
        if ((brace_level = 1) ∧ (ex_buf_ptr + 1 < ex_buf_length)) then
          if (ex_buf[ex_buf_ptr + 1] = backslash) then ⟨ Purify a special character 432 ⟩;
        end
      else if (ex_buf[ex_buf_ptr] = right_brace) then
          if (brace_level > 0) then decr(brace_level)
    endcases; incr(ex_buf_ptr);
    end;
  ex_buf_length ← ex_buf_xptr;
  end
```

This code is used in section 430.

432. Special characters (even without a matching *right_brace*) are purified by removing the control sequences (but restoring the correct thing for '`\i`' and '`\j`' as well as the eleven alphabetic foreign characters in Table 3.2 of the LaTeX manual) and removing all nonalphanumeric characters (including *white_space* and *sep_chars*).

```
⟨ Purify a special character 432 ⟩ ≡
  begin incr(ex_buf_ptr); { skip over the left_brace }
  while ((ex_buf_ptr < ex_buf_length) ∧ (brace_level > 0)) do
    begin incr(ex_buf_ptr); { skip over the backslash }
    ex_buf_yptr ← ex_buf_ptr; { mark the beginning of the control sequence }
    while ((ex_buf_ptr < ex_buf_length) ∧ (lex_class[ex_buf[ex_buf_ptr]] = alpha)) do
      incr(ex_buf_ptr); { this scans the control sequence }
    control_seq_loc ← str_lookup(ex_buf, ex_buf_yptr, ex_buf_ptr − ex_buf_yptr, control_seq_ilk, dont_insert);
    if (hash_found) then ⟨ Purify this accented or foreign character 433 ⟩;
    while ((ex_buf_ptr < ex_buf_length) ∧ (brace_level > 0) ∧ (ex_buf[ex_buf_ptr] ≠ backslash)) do
      begin { this scans to the next control sequence }
      case (lex_class[ex_buf[ex_buf_ptr]]) of
      alpha, numeric: begin ex_buf[ex_buf_xptr] ← ex_buf[ex_buf_ptr]; incr(ex_buf_xptr);
        end;
      othercases if (ex_buf[ex_buf_ptr] = right_brace) then decr(brace_level)
        else if (ex_buf[ex_buf_ptr] = left_brace) then incr(brace_level)
      endcases; incr(ex_buf_ptr);
      end;
    end;
  decr(ex_buf_ptr); { unskip the right_brace (or last character) }
  end
```

This code is used in section 431.

433. We consider the purified character to be either the first alphabetic character of its control sequence, or perhaps both alphabetic characters.

```
⟨ Purify this accented or foreign character 433 ⟩ ≡
  begin ex_buf[ex_buf_xptr] ← ex_buf[ex_buf_yptr]; { the first alphabetic character }
  incr(ex_buf_xptr);
  case (ilk_info[control_seq_loc]) of
  n_oe, n_oe_upper, n_ae, n_ae_upper, n_ss: begin { and the second }
    ex_buf[ex_buf_xptr] ← ex_buf[ex_buf_yptr + 1]; incr(ex_buf_xptr);
    end;
  othercases do_nothing
  endcases;
  end
```

This code is used in section 432.

434. The *built_in* function `quote$` pushes the string consisting of the *double_quote* character.

```
⟨ execute_fn(quote$) 434 ⟩ ≡
procedure x_quote;
  begin str_room(1); append_char(double_quote); push_lit_stk(make_string, stk_str);
  end;
```

This code is used in section 342.

435. The *built_in* function `skip$` is a no-op.

⟨ *execute_fn*(`skip$`) 435 ⟩ ≡
```
begin do_nothing;
end
```
This code is used in section 341.

436. The *built_in* function `stack$` pops and prints the whole stack; it's meant to be used for style designers while debugging.

⟨ *execute_fn*(`stack$`) 436 ⟩ ≡
```
begin pop_whole_stack;
end
```
This code is used in section 341.

437. The *built_in* function `substring$` pops the top three literals (they are the two integers literals *pop_lit1* and *pop_lit2* and a string literal, in that order). It pushes the substring of the (at most) *pop_lit1* consecutive characters starting at the *pop_lit2*th character (assuming 1-based indexing) if *pop_lit2* is positive, and ending at the $-$*pop_lit2*th character from the end if *pop_lit2* is negative (where the first character from the end is the last character). If any of the types is incorrect, it complain and pushes the null string.

⟨ *execute_fn*(`substring$`) 437 ⟩ ≡
```
procedure x_substring;
  label exit;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2); pop_lit_stk(pop_lit3, pop_typ3);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(s_null, stk_str);
    end
  else if (pop_typ2 ≠ stk_int) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_int); push_lit_stk(s_null, stk_str);
      end
    else if (pop_typ3 ≠ stk_str) then
        begin print_wrong_stk_lit(pop_lit3, pop_typ3, stk_str); push_lit_stk(s_null, stk_str);
        end
      else begin sp_length ← length(pop_lit3);
        if (pop_lit1 ≥ sp_length) then
          if ((pop_lit2 = 1) ∨ (pop_lit2 = −1)) then
            begin repush_string; return;
            end;
        if ((pop_lit1 ≤ 0) ∨ (pop_lit2 = 0) ∨ (pop_lit2 > sp_length) ∨ (pop_lit2 < −sp_length)) then
          begin push_lit_stk(s_null, stk_str); return;
          end
        else ⟨ Form the appropriate substring 438 ⟩;
        end;
exit: end;
```
This code is used in section 342.

438. This module finds the substring as described in the last section, and slides it into place in the string pool, if necessary.

```
⟨ Form the appropriate substring 438 ⟩ ≡
  begin if (pop_lit2 > 0) then
    begin if (pop_lit1 > sp_length − (pop_lit2 − 1)) then pop_lit1 ← sp_length − (pop_lit2 − 1);
    sp_ptr ← str_start[pop_lit3] + (pop_lit2 − 1); sp_end ← sp_ptr + pop_lit1;
    if (pop_lit2 = 1) then
      if (pop_lit3 ≥ cmd_str_ptr) then { no shifting—merely change pointers }
        begin str_start[pop_lit3 + 1] ← sp_end; unflush_string; incr(lit_stk_ptr); return;
        end;
    end
  else { −ex_buf_length ≤ pop_lit2 < 0 }
  begin pop_lit2 ← −pop_lit2;
  if (pop_lit1 > sp_length − (pop_lit2 − 1)) then pop_lit1 ← sp_length − (pop_lit2 − 1);
  sp_end ← str_start[pop_lit3 + 1] − (pop_lit2 − 1); sp_ptr ← sp_end − pop_lit1;
  end;
  while (sp_ptr < sp_end) do { shift the substring }
    begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
    end;
  push_lit_stk(make_string, stk_str); { and push it onto the stack }
  end
```

This code is used in section 437.

439. The *built_in* function `swap$` pops the top two literals from the stack and pushes them back swapped.

```
⟨ execute_fn(swap$) 439 ⟩ ≡
procedure x_swap;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if ((pop_typ1 ≠ stk_str) ∨ (pop_lit1 < cmd_str_ptr)) then
    begin push_lit_stk(pop_lit1, pop_typ1);
    if ((pop_typ2 = stk_str) ∧ (pop_lit2 ≥ cmd_str_ptr)) then unflush_string;
    push_lit_stk(pop_lit2, pop_typ2);
    end
  else if ((pop_typ2 ≠ stk_str) ∨ (pop_lit2 < cmd_str_ptr)) then
      begin unflush_string; { this is pop_lit1 }
      push_lit_stk(pop_lit1, stk_str); push_lit_stk(pop_lit2, pop_typ2);
      end
    else { bummer, both are recent strings }
  ⟨ Swap the two strings (they're at the end of str_pool) 440 ⟩;
  end;
```

This code is used in section 342.

440. We have to swap both (a) the strings at the end of the string pool, and (b) their pointers on the literal stack.

```
⟨ Swap the two strings (they're at the end of str_pool) 440 ⟩ ≡
  begin ex_buf_length ← 0; add_buf_pool(pop_lit2); { save the second string }
  sp_ptr ← str_start[pop_lit1]; sp_end ← str_start[pop_lit1 + 1];
  while (sp_ptr < sp_end) do { slide the first string down }
    begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
    end;
  push_lit_stk(make_string, stk_str); { and push it onto the stack }
  add_pool_buf_and_push; { push second string onto the stack }
  end
```

This code is used in section 439.

441. The *built_in* function `text.length$` pops the top (string) literal, and pushes the number of text characters it contains, where an accented character (more precisely, a "special character", defined earlier) counts as a single text character, even if it's missing its matching *right_brace*, and where braces don't count as text characters. If the literal isn't a string, it complains and pushes the null string.

```
⟨ execute_fn(text.length$) 441 ⟩ ≡
procedure x_text_length;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(s_null, stk_str);
    end
  else begin num_text_chars ← 0; ⟨ Count the text characters 442 ⟩;
    push_lit_stk(num_text_chars, stk_int); { and push it onto the stack }
    end;
  end;
```

This code is used in section 342.

442. Here we determine the number of text characters in the string, where an entire special character counts as a single text character (even if it's missing its matching *right_brace*), and where braces don't count as text characters.

```
⟨ Count the text characters 442 ⟩ ≡
  begin sp_ptr ← str_start[pop_lit1]; sp_end ← str_start[pop_lit1 + 1]; sp_brace_level ← 0;
  while (sp_ptr < sp_end) do
    begin incr(sp_ptr);
    if (str_pool[sp_ptr − 1] = left_brace) then
      begin incr(sp_brace_level);
      if ((sp_brace_level = 1) ∧ (sp_ptr < sp_end)) then
        if (str_pool[sp_ptr] = backslash) then
          begin incr(sp_ptr);   { skip over the backslash }
          while ((sp_ptr < sp_end) ∧ (sp_brace_level > 0)) do
            begin if (str_pool[sp_ptr] = right_brace) then decr(sp_brace_level)
            else if (str_pool[sp_ptr] = left_brace) then incr(sp_brace_level);
            incr(sp_ptr);
            end;
          incr(num_text_chars);
          end;
      end
    else if (str_pool[sp_ptr − 1] = right_brace) then
        begin if (sp_brace_level > 0) then decr(sp_brace_level);
        end
      else incr(num_text_chars);
    end;
  end
```

This code is used in section 441.

443. The *built_in* function `text.prefix$` pops the top two literals (the integer literal *pop_lit1* and a string literal, in that order). It pushes the substring of the (at most) *pop_lit1* consecutive text characters starting from the beginning of the string. This function is similar to `substring$`, but this one considers an accented character (or more precisely, a "special character", even if it's missing its matching *right_brace*) to be a single text character (rather than however many *ASCII_code* characters it actually comprises), and this function doesn't consider braces to be text characters; furthermore, this function appends any needed matching *right_brace*s. If any of the types is incorrect, it complains and pushes the null string.

```
⟨ execute_fn(text.prefix$) 443 ⟩ ≡
procedure x_text_prefix;
  label exit;
  begin pop_lit_stk(pop_lit1, pop_typ1); pop_lit_stk(pop_lit2, pop_typ2);
  if (pop_typ1 ≠ stk_int) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); push_lit_stk(s_null, stk_str);
    end
  else if (pop_typ2 ≠ stk_str) then
      begin print_wrong_stk_lit(pop_lit2, pop_typ2, stk_str); push_lit_stk(s_null, stk_str);
      end
    else if (pop_lit1 ≤ 0) then
        begin push_lit_stk(s_null, stk_str); return;
        end
      else ⟨ Form the appropriate prefix 444 ⟩;
exit: end;
```

This code is used in section 342.

444. This module finds the prefix as described in the last section, and appends any needed matching *right_brace*s.

```
⟨ Form the appropriate prefix 444 ⟩ ≡
  begin sp_ptr ← str_start[pop_lit2]; sp_end ← str_start[pop_lit2 + 1]; { this may change }
  ⟨ Scan the appropriate number of characters 445 ⟩;
  if (pop_lit2 ≥ cmd_str_ptr) then { no shifting—merely change pointers }
    pool_ptr ← sp_end
  else while (sp_ptr < sp_end) do { shift the substring }
      begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
      end;
  while (sp_brace_level > 0) do { add matching right_braces }
    begin append_char(right_brace); decr(sp_brace_level);
    end;
  push_lit_stk(make_string, stk_str); { and push it onto the stack }
  end
```

This code is used in section 443.

445. This section scans *pop_lit1* text characters, where an entire special character counts as a single text character (even if it's missing its matching *right_brace*), and where braces don't count as text characters.

```
⟨ Scan the appropriate number of characters 445 ⟩ ≡
  begin num_text_chars ← 0; sp_brace_level ← 0; sp_xptr1 ← sp_ptr;
  while ((sp_xptr1 < sp_end) ∧ (num_text_chars < pop_lit1)) do
    begin incr(sp_xptr1);
    if (str_pool[sp_xptr1 − 1] = left_brace) then
      begin incr(sp_brace_level);
      if ((sp_brace_level = 1) ∧ (sp_xptr1 < sp_end)) then
        if (str_pool[sp_xptr1] = backslash) then
          begin incr(sp_xptr1); { skip over the backslash }
          while ((sp_xptr1 < sp_end) ∧ (sp_brace_level > 0)) do
            begin if (str_pool[sp_xptr1] = right_brace) then decr(sp_brace_level)
            else if (str_pool[sp_xptr1] = left_brace) then incr(sp_brace_level);
            incr(sp_xptr1);
            end;
          incr(num_text_chars);
          end;
      end
    else if (str_pool[sp_xptr1 − 1] = right_brace) then
        begin if (sp_brace_level > 0) then decr(sp_brace_level);
        end
      else incr(num_text_chars);
    end;
  sp_end ← sp_xptr1;
  end
```

This code is used in section 444.

446. The *built_in* function `top$` pops and prints the top of the stack.

```
⟨ execute_fn(top$) 446 ⟩ ≡
  begin pop_top_and_print;
  end
```

This code is used in section 341.

447. The *built_in* function `type$` pushes the appropriate string from *type_list* onto the stack (unless either it's *undefined* or *empty*, in which case it pushes the null string).

```
⟨ execute_fn(type$) 447 ⟩ ≡
procedure x_type;
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else if ((type_list[cite_ptr] = undefined) ∨ (type_list[cite_ptr] = empty)) then push_lit_stk(s_null, stk_str)
    else push_lit_stk(hash_text[type_list[cite_ptr]], stk_str);
  end;
```

This code is used in section 342.

448. The *built_in* function `warning$` pops the top (string) literal and prints it following a warning message. This is implemented as a special *built_in* function rather than using the `top$` function so that it can *mark_warning*.

```
⟨ execute_fn(warning$) 448 ⟩ ≡
procedure x_warning;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str)
  else begin print('Warning--'); print_lit(pop_lit1, pop_typ1); mark_warning;
    end;
  end;
```

This code is used in section 342.

449. The *built_in* function `while$` pops the top two (function) literals, and keeps executing the second as long as the (integer) value left on the stack by executing the first is greater than 0. If either type is incorrect, it complains but does nothing else.

```
⟨ execute_fn(while$) 449 ⟩ ≡
  begin pop_lit_stk(r_pop_lt1, r_pop_tp1); pop_lit_stk(r_pop_lt2, r_pop_tp2);
  if (r_pop_tp1 ≠ stk_fn) then print_wrong_stk_lit(r_pop_lt1, r_pop_tp1, stk_fn)
  else if (r_pop_tp2 ≠ stk_fn) then print_wrong_stk_lit(r_pop_lt2, r_pop_tp2, stk_fn)
    else loop
        begin execute_fn(r_pop_lt2);  { this is the while$ test }
        pop_lit_stk(pop_lit1, pop_typ1);
        if (pop_typ1 ≠ stk_int) then
          begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_int); goto end_while;
          end
        else if (pop_lit1 > 0) then execute_fn(r_pop_lt1)  { this is the while$ body }
          else goto end_while;
        end;
end_while:  { justifies this mean_while }
  end
```

This code is used in section 341.

450. The *built_in* function `width$` pops the top (string) literal and pushes the integer that represents its width in units specified by the *char_width* array. This function takes the literal literally; that is, it assumes each character in the string is to be printed as is, regardless of whether the character has a special meaning to TEX, except that special characters (even without their *right_brace*s) are handled specially. If the literal isn't a string, it complains and pushes 0.

```
⟨ execute_fn(width$) 450 ⟩ ≡
procedure x_width;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then
    begin print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str); push_lit_stk(0, stk_int);
    end
  else begin ex_buf_length ← 0; add_buf_pool(pop_lit1); string_width ← 0;
    ⟨ Add up the char_widths in this string 451 ⟩;
    push_lit_stk(string_width, stk_int);
    end
  end;
```

This code is used in section 342.

451. We use the natural width for all but special characters, and we complain if the string isn't brace-balanced.

```
⟨ Add up the char_widths in this string 451 ⟩ ≡
  begin brace_level ← 0;  { we're at the top level }
  ex_buf_ptr ← 0;  { and the beginning of string }
  while (ex_buf_ptr < ex_buf_length) do
    begin if (ex_buf[ex_buf_ptr] = left_brace) then
      begin incr(brace_level);
      if ((brace_level = 1) ∧ (ex_buf_ptr + 1 < ex_buf_length)) then
        if (ex_buf[ex_buf_ptr + 1] = backslash) then ⟨ Determine the width of this special character 452 ⟩
        else string_width ← string_width + char_width[left_brace]
      else string_width ← string_width + char_width[left_brace];
      end
    else if (ex_buf[ex_buf_ptr] = right_brace) then
        begin decr_brace_level(pop_lit1); string_width ← string_width + char_width[right_brace];
        end
      else string_width ← string_width + char_width[ex_buf[ex_buf_ptr]];
    incr(ex_buf_ptr);
    end;
  check_brace_level(pop_lit1);
  end
```

This code is used in section 450.

452. We use the natural widths of all characters except that some characters have no width: braces, control sequences (except for the usual 13 accented and foreign characters, whose widths are given in the next module), and *white_space* following control sequences (even a null control sequence).

```
⟨ Determine the width of this special character 452 ⟩ ≡
  begin incr(ex_buf_ptr); { skip over the left_brace }
  while ((ex_buf_ptr < ex_buf_length) ∧ (brace_level > 0)) do
    begin incr(ex_buf_ptr); { skip over the backslash }
    ex_buf_xptr ← ex_buf_ptr;
    while ((ex_buf_ptr < ex_buf_length) ∧ (lex_class[ex_buf[ex_buf_ptr]] = alpha)) do
      incr(ex_buf_ptr); { this scans the control sequence }
    if ((ex_buf_ptr < ex_buf_length) ∧ (ex_buf_ptr = ex_buf_xptr)) then incr(ex_buf_ptr)
          { this skips a nonalpha control seq }
    else begin control_seq_loc ← str_lookup(ex_buf, ex_buf_xptr, ex_buf_ptr − ex_buf_xptr, control_seq_ilk,
          dont_insert);
      if (hash_found) then ⟨ Determine the width of this accented or foreign character 453 ⟩;
      end;
    while ((ex_buf_ptr < ex_buf_length) ∧ (lex_class[ex_buf[ex_buf_ptr]] = white_space)) do
      incr(ex_buf_ptr); { this skips following white_space }
    while ((ex_buf_ptr < ex_buf_length) ∧ (brace_level > 0) ∧ (ex_buf[ex_buf_ptr] ≠ backslash)) do
      begin { this scans to the next control sequence }
      if (ex_buf[ex_buf_ptr] = right_brace) then decr(brace_level)
      else if (ex_buf[ex_buf_ptr] = left_brace) then incr(brace_level)
        else string_width ← string_width + char_width[ex_buf[ex_buf_ptr]];
      incr(ex_buf_ptr);
      end;
    end;
  decr(ex_buf_ptr); { unskip the right_brace }
  end
```

This code is used in section 451.

453. Five of the 13 possibilities resort to special information not present in the *char_width* array; the other eight simply use *char_width*'s information for the first letter of the control sequence.

```
⟨ Determine the width of this accented or foreign character 453 ⟩ ≡
  begin case (ilk_info[control_seq_loc]) of
  n_ss: string_width ← string_width + ss_width;
  n_ae: string_width ← string_width + ae_width;
  n_oe: string_width ← string_width + oe_width;
  n_ae_upper: string_width ← string_width + upper_ae_width;
  n_oe_upper: string_width ← string_width + upper_oe_width;
  othercases string_width ← string_width + char_width[ex_buf[ex_buf_xptr]]
  endcases;
  end
```

This code is used in section 452.

454. The *built_in* function `write$` pops the top (string) literal and writes it onto the output buffer *out_buf* (which will result in stuff being written onto the `.bbl` file if the buffer fills up). If the literal isn't a string, it complains but does nothing else.

⟨ *execute_fn*(`write$`) 454 ⟩ ≡
```
procedure x_write;
  begin pop_lit_stk(pop_lit1, pop_typ1);
  if (pop_typ1 ≠ stk_str) then print_wrong_stk_lit(pop_lit1, pop_typ1, stk_str)
  else add_out_pool(pop_lit1);
  end;
```
This code is used in section 342.

455. Cleaning up. This section does any last-minute printing and ends the program.

⟨ Clean up and leave 455 ⟩ ≡
```
begin if ((read_performed) ∧ (¬reading_completed)) then
  begin print(´Aborted␣at␣line␣´, bib_line_num : 0, ´␣of␣file␣´); print_bib_name;
  end;
trace_and_stat_printing; ⟨ Print the job history 466 ⟩;
a_close(log_file); { turn out the lights, the fat lady has sung; it's over, Yogi }
end
```
This code is used in section 10.

456. Here we print **trace** and/or **stat** information, if desired.

⟨ Procedures and functions for all file I/O, error messages, and such 3 ⟩ +≡
```
procedure trace_and_stat_printing;
  begin trace ⟨ Print all .bib- and .bst-file information 457 ⟩;
  ⟨ Print all cite_list and entry information 458 ⟩;
  ⟨ Print the wiz_defined functions 463 ⟩;
  ⟨ Print the string pool 464 ⟩;
  ecart
  stat ⟨ Print usage statistics 465 ⟩;
  tats
  end;
```

457. This prints information obtained from the `.aux` file about the other files.

⟨ Print all .bib- and .bst-file information 457 ⟩ ≡
```
begin if (num_bib_files = 1) then trace_pr_ln(´The␣1␣database␣file␣is´)
else trace_pr_ln(´The␣´, num_bib_files : 0, ´␣database␣files␣are´);
if (num_bib_files = 0) then trace_pr_ln(´␣␣␣undefined´)
else begin bib_ptr ← 0;
  while (bib_ptr < num_bib_files) do
    begin trace_pr(´␣␣␣´); trace_pr_pool_str(cur_bib_str); trace_pr_pool_str(s_bib_extension);
    trace_pr_newline; incr(bib_ptr);
    end;
  end;
trace_pr(´The␣style␣file␣is␣´);
if (bst_str = 0) then trace_pr_ln(´undefined´)
else begin trace_pr_pool_str(bst_str); trace_pr_pool_str(s_bst_extension); trace_pr_newline;
  end;
end
```
This code is used in section 456.

458. In entry-sorted order, this prints an entry's *cite_list* string and, indirectly, its entry type and entry variables.

```
⟨ Print all cite_list and entry information 458 ⟩ ≡
  begin if (all_entries) then trace_pr(´all_marker=´, all_marker : 0, ´,␣´);
  if (read_performed) then trace_pr_ln(´old_num_cites=´, old_num_cites : 0)
  else trace_pr_newline;
  trace_pr(´The␣´, num_cites : 0);
  if (num_cites = 1) then trace_pr_ln(´␣entry:´)
  else trace_pr_ln(´␣entries:´);
  if (num_cites = 0) then trace_pr_ln(´␣␣␣undefined´)
  else begin sort_cite_ptr ← 0;
    while (sort_cite_ptr < num_cites) do
      begin if (¬read_completed) then { we didn't finish the read command }
        cite_ptr ← sort_cite_ptr
      else cite_ptr ← sorted_cites[sort_cite_ptr];
      trace_pr_pool_str(cur_cite_str);
      if (read_performed) then ⟨ Print entry information 459 ⟩
      else trace_pr_newline;
      incr(sort_cite_ptr);
      end;
    end;
  end
```

This code is used in section 456.

459. This prints information gathered while reading the `.bst` and `.bib` files.

```
⟨ Print entry information 459 ⟩ ≡
  begin trace_pr(´,␣entry-type␣´);
  if (type_list[cite_ptr] = undefined) then
  undefined: trace_pr(´unknown´)
  else if (type_list[cite_ptr] = empty) then trace_pr(´---␣no␣type␣found´)
    else trace_pr_pool_str(hash_text[type_list[cite_ptr]]);
  trace_pr_ln(´,␣has␣entry␣strings´); ⟨ Print entry strings 460 ⟩;
  trace_pr(´␣␣has␣entry␣integers´); ⟨ Print entry integers 461 ⟩;
  trace_pr_ln(´␣␣and␣has␣fields´); ⟨ Print fields 462 ⟩;
  end
```

This code is used in section 458.

460. This prints, for the current entry, the strings declared by the `entry` command.

```
⟨ Print entry strings 460 ⟩ ≡
  begin if (num_ent_strs = 0) then trace_pr_ln('␣␣␣␣undefined')
  else if (¬read_completed) then trace_pr_ln('␣␣␣␣uninitialized')
    else begin str_ent_ptr ← cite_ptr * num_ent_strs;
      while (str_ent_ptr < (cite_ptr + 1) * num_ent_strs) do
        begin ent_chr_ptr ← 0; trace_pr('␣␣␣␣"');
        while (entry_strs[str_ent_ptr][ent_chr_ptr] ≠ end_of_string) do
          begin trace_pr(xchr[entry_strs[str_ent_ptr][ent_chr_ptr]]); incr(ent_chr_ptr);
          end;
        trace_pr_ln('"'); incr(str_ent_ptr);
        end;
      end;
  end
```

This code is used in section 459.

461. This prints, for the current entry, the integers declared by the `entry` command.

```
⟨ Print entry integers 461 ⟩ ≡
  begin if (num_ent_ints = 0) then trace_pr('␣undefined')
  else if (¬read_completed) then trace_pr('␣uninitialized')
    else begin int_ent_ptr ← cite_ptr * num_ent_ints;
      while (int_ent_ptr < (cite_ptr + 1) * num_ent_ints) do
        begin trace_pr('␣', entry_ints[int_ent_ptr] : 0); incr(int_ent_ptr);
        end;
      end;
  trace_pr_newline;
  end
```

This code is used in section 459.

462. This prints the fields stored for the current entry.

```
⟨ Print fields 462 ⟩ ≡
  begin if (¬read_performed) then trace_pr_ln('␣␣␣␣uninitialized')
  else begin field_ptr ← cite_ptr * num_fields; field_end_ptr ← field_ptr + num_fields; no_fields ← true;
    while (field_ptr < field_end_ptr) do
      begin if (field_info[field_ptr] ≠ missing) then
        begin trace_pr('␣␣␣␣"'); trace_pr_pool_str(field_info[field_ptr]); trace_pr_ln('"');
        no_fields ← false;
        end;
      incr(field_ptr);
      end;
    if (no_fields) then trace_pr_ln('␣␣␣␣missing');
    end;
  end
```

This code is used in section 459.

463. This gives all the *wiz_defined* functions that appeared in the `.bst` file.

```
⟨ Print the wiz_defined functions 463 ⟩ ≡
  begin trace_pr_ln(´The␣wiz-defined␣functions␣are´);
  if (wiz_def_ptr = 0) then trace_pr_ln(´␣␣␣nonexistent´)
  else begin wiz_fn_ptr ← 0;
    while (wiz_fn_ptr < wiz_def_ptr) do
      begin if (wiz_functions[wiz_fn_ptr] = end_of_def) then
        trace_pr_ln(wiz_fn_ptr : 0, ´--end-of-def--´)
      else if (wiz_functions[wiz_fn_ptr] = quote_next_fn) then
          trace_pr(wiz_fn_ptr : 0, ´␣␣quote_next_function␣␣␣␣´)
        else begin trace_pr(wiz_fn_ptr : 0, ´␣␣`´); trace_pr_pool_str(hash_text[wiz_functions[wiz_fn_ptr]]);
          trace_pr_ln(´´´´);
          end;
      incr(wiz_fn_ptr);
      end;
    end;
  end
```

This code is used in section 456.

464. This includes all the 'static' strings (that is, those that are also in the hash table), but none of the dynamic strings (that is, those put on the stack while executing `.bst` commands).

```
⟨ Print the string pool 464 ⟩ ≡
  begin trace_pr_ln(´The␣string␣pool␣is´); str_num ← 1;
  while (str_num < str_ptr) do
    begin trace_pr(str_num : 4, str_start[str_num] : 6, ´␣"´); trace_pr_pool_str(str_num); trace_pr_ln(´"´);
    incr(str_num);
    end;
  end
```

This code is used in section 456.

465. These statistics can help determine how large some of the constants should be and can tell how useful certain *built_in* functions are. They are written to the same files as tracing information.

```
define stat_pr ≡ trace_pr
define stat_pr_ln ≡ trace_pr_ln
define stat_pr_pool_str ≡ trace_pr_pool_str
```

⟨ Print usage statistics 465 ⟩ ≡

```
begin stat_pr(´You´´ve␣used␣´, num_cites : 0);
if (num_cites = 1) then stat_pr_ln(´␣entry,´)
else stat_pr_ln(´␣entries,´);
stat_pr_ln(´␣␣␣␣␣␣␣␣␣␣␣´, wiz_def_ptr : 0, ´␣wiz_defined-function␣locations,´);
stat_pr_ln(´␣␣␣␣␣␣␣␣␣␣␣´, str_ptr : 0, ´␣strings␣with␣´, str_start[str_ptr] : 0, ´␣characters,´);
blt_in_ptr ← 0; total_ex_count ← 0;
while (blt_in_ptr < num_blt_in_fns) do
  begin total_ex_count ← total_ex_count + execution_count[blt_in_ptr]; incr(blt_in_ptr);
  end;
stat_pr_ln(´and␣the␣built_in␣function-call␣counts,␣´, total_ex_count : 0, ´␣in␣all,␣are:´);
blt_in_ptr ← 0;
while (blt_in_ptr < num_blt_in_fns) do
  begin stat_pr_pool_str(hash_text[blt_in_loc[blt_in_ptr]]);
  stat_pr_ln(´␣--␣´, execution_count[blt_in_ptr] : 0); incr(blt_in_ptr);
  end;
end
```

This code is used in section 456.

466. Some implementations may wish to pass the *history* value to the operating system so that it can be used to govern whether or not other programs are started. Here we simply report the history to the user.

⟨ Print the job *history* 466 ⟩ ≡

```
case (history) of
spotless: do_nothing;
warning_message: begin if (err_count = 1) then print_ln(´(There␣was␣1␣warning)´)
  else print_ln(´(There␣were␣´, err_count : 0, ´␣warnings)´);
  end;
error_message: begin if (err_count = 1) then print_ln(´(There␣was␣1␣error␣message)´)
  else print_ln(´(There␣were␣´, err_count : 0, ´␣error␣messages)´);
  end;
fatal_message: print_ln(´(That␣was␣a␣fatal␣error)´);
othercases begin print(´History␣is␣bunk´); print_confusion;
  end
endcases
```

This code is used in section 455.

467. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make BibTeX work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

468. Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. All references are to section numbers instead of page numbers.

This index also lists a few error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing TEX in a new operating environment. A list of various things that can't happen appears under "this can't happen".

⟨ Add cross-reference information 277 ⟩ Used in section 276.
⟨ Add extensions and open files 106 ⟩ Used in section 103.
⟨ Add or update a cross reference on *cite_list* if necessary 264 ⟩ Used in section 263.
⟨ Add the *period* (it's necessary) and push 362 ⟩ Used in section 361.
⟨ Add the *period*, if necessary, and push 361 ⟩ Used in section 360.
⟨ Add up the *char_width*s in this string 451 ⟩ Used in section 450.
⟨ Assign to a *str_entry_var* 357 ⟩ Used in section 354.
⟨ Assign to a *str_global_var* 359 ⟩ Used in section 354.
⟨ Assign to an *int_entry_var* 355 ⟩ Used in section 354.
⟨ Assign to an *int_global_var* 358 ⟩ Used in section 354.
⟨ Break that line 323 ⟩ Used in section 322.
⟨ Break that unbreakably long line 324 ⟩ Used in section 323.
⟨ Check and insert the quoted function 193 ⟩ Used in section 192.
⟨ Check for a database key of interest 267 ⟩ Used in section 266.
⟨ Check for a duplicate or `crossref`-matching database key 268 ⟩ Used in section 267.
⟨ Check for entire database inclusion (and thus skip this cite key) 134 ⟩ Used in section 133.
⟨ Check the `execute`-command argument token 179 ⟩ Used in section 178.
⟨ Check the `iterate`-command argument token 204 ⟩ Used in section 203.
⟨ Check the `reverse`-command argument token 213 ⟩ Used in section 212.
⟨ Check the "constant" values for consistency 17, 302 ⟩ Used in section 13.
⟨ Check the cite key 133 ⟩ Used in section 132.
⟨ Check the macro name 207 ⟩ Used in section 206.
⟨ Check the special character (and **return**) 398 ⟩ Used in section 397.
⟨ Check the *wiz_defined* function name 182 ⟩ Used in section 181.
⟨ Cite seen, don't add a cite key 135 ⟩ Used in section 133.
⟨ Cite unseen, add a cite key 136 ⟩ Used in section 133.
⟨ Clean up and leave 455 ⟩ Used in section 10.
⟨ Compiler directives 11 ⟩ Used in section 10.
⟨ Complain about a nested cross reference 282 ⟩ Used in section 279.
⟨ Complain about missing entries whose cite keys got overwritten 286 ⟩ Used in section 283.
⟨ Complete this function's definition 200 ⟩ Used in section 187.
⟨ Compute the hash code h 69 ⟩ Used in section 68.
⟨ Concatenate the two strings and push 351 ⟩ Used in section 350.
⟨ Concatenate them and push when $pop\_lit1, pop\_lit2 < cmd\_str\_ptr$ 353 ⟩ Used in section 352.
⟨ Concatenate them and push when $pop\_lit2 < cmd\_str\_ptr$ 352 ⟩ Used in section 351.
⟨ Constants in the outer block 14, 333 ⟩ Used in section 10.
⟨ Convert a noncontrol sequence 375 ⟩ Used in section 371.
⟨ Convert a special character 371 ⟩ Used in section 370.
⟨ Convert a $brace\_level = 0$ character 376 ⟩ Used in section 370.
⟨ Convert the accented or foreign character, if necessary 372 ⟩ Used in section 371.
⟨ Convert, then remove the control sequence 374 ⟩ Used in section 372.
⟨ Copy name and count *comma*s to determine syntax 387 ⟩ Used in section 382.
⟨ Copy the macro string to *field_vl_str* 260 ⟩ Used in section 259.
⟨ Count the text characters 442 ⟩ Used in section 441.
⟨ Declarations for executing *built_in* functions 343 ⟩ Used in section 325.
⟨ Determine the case-conversion type 366 ⟩ Used in section 364.
⟨ Determine the number of names 427 ⟩ Used in section 426.
⟨ Determine the width of this accented or foreign character 453 ⟩ Used in section 452.
⟨ Determine the width of this special character 452 ⟩ Used in section 451.
⟨ Determine where the first name ends and von name starts and ends 396 ⟩ Used in section 395.
⟨ Do a full brace-balanced scan 256 ⟩ Used in section 253.
⟨ Do a full scan with $bib\_brace\_level > 0$ 257 ⟩ Used in section 256.

⟨ *execute_fn* itself 325 ⟩ Used in section 342.
⟨ *execute_fn*(`*`) 350 ⟩ Used in section 342.
⟨ *execute_fn*(`+`) 348 ⟩ Used in section 342.
⟨ *execute_fn*(`-`) 349 ⟩ Used in section 342.
⟨ *execute_fn*(`:=`) 354 ⟩ Used in section 342.
⟨ *execute_fn*(`<`) 347 ⟩ Used in section 342.
⟨ *execute_fn*(`=`) 345 ⟩ Used in section 342.
⟨ *execute_fn*(`>`) 346 ⟩ Used in section 342.
⟨ *execute_fn*(`add.period$`) 360 ⟩ Used in section 342.
⟨ *execute_fn*(`call.type$`) 363 ⟩ Used in section 341.
⟨ *execute_fn*(`change.case$`) 364 ⟩ Used in section 342.
⟨ *execute_fn*(`chr.to.int$`) 377 ⟩ Used in section 342.
⟨ *execute_fn*(`cite$`) 378 ⟩ Used in section 342.
⟨ *execute_fn*(`duplicate$`) 379 ⟩ Used in section 342.
⟨ *execute_fn*(`empty$`) 380 ⟩ Used in section 342.
⟨ *execute_fn*(`format.name$`) 382 ⟩ Used in section 342.
⟨ *execute_fn*(`if$`) 421 ⟩ Used in section 341.
⟨ *execute_fn*(`int.to.chr$`) 422 ⟩ Used in section 342.
⟨ *execute_fn*(`int.to.str$`) 423 ⟩ Used in section 342.
⟨ *execute_fn*(`missing$`) 424 ⟩ Used in section 342.
⟨ *execute_fn*(`newline$`) 425 ⟩ Used in section 341.
⟨ *execute_fn*(`num.names$`) 426 ⟩ Used in section 342.
⟨ *execute_fn*(`pop$`) 428 ⟩ Used in section 341.
⟨ *execute_fn*(`preamble$`) 429 ⟩ Used in section 342.
⟨ *execute_fn*(`purify$`) 430 ⟩ Used in section 342.
⟨ *execute_fn*(`quote$`) 434 ⟩ Used in section 342.
⟨ *execute_fn*(`skip$`) 435 ⟩ Used in section 341.
⟨ *execute_fn*(`stack$`) 436 ⟩ Used in section 341.
⟨ *execute_fn*(`substring$`) 437 ⟩ Used in section 342.
⟨ *execute_fn*(`swap$`) 439 ⟩ Used in section 342.
⟨ *execute_fn*(`text.length$`) 441 ⟩ Used in section 342.
⟨ *execute_fn*(`text.prefix$`) 443 ⟩ Used in section 342.
⟨ *execute_fn*(`top$`) 446 ⟩ Used in section 341.
⟨ *execute_fn*(`type$`) 447 ⟩ Used in section 342.
⟨ *execute_fn*(`warning$`) 448 ⟩ Used in section 342.
⟨ *execute_fn*(`while$`) 449 ⟩ Used in section 341.
⟨ *execute_fn*(`width$`) 450 ⟩ Used in section 342.
⟨ *execute_fn*(`write$`) 454 ⟩ Used in section 342.

The BibTeX preprocessor

(Version 0.99d—January 12, 2013)